Task: UR20
CDRL: 01000

UR20--Process/Environment Integration Ada/Xt Architecture: Design Report

Informal Technical Data

UNISYS

STARS-RC-01000/001/00
25 January 1990

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	25 January 1990	Final

4. TITLE AND SUBTITLE
Ada/Xt Architecture: Design Report

5. FUNDING NUMBERS
STARS Contract
F19628-88-D-0031

6. AUTHOR(S)
Kurt Wallnau

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091

8. PERFORMING ORGANIZATION REPORT NUMBER
GR-7670-1107(NP)

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Department of the Air Force
Headquarters, Electronic Systems Division (AFSC)
Hanscom AFB, MA 01731-5000

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
01000

11. SUPPLEMENTARY NOTES
This report describes the design of the Process/Environment Integration Ada/XT Toolkit, SunOS Implementation

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release;
distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report provides a detailed description of the Ada/Xt toolkit architecture. The purpose of this report is to describe the Ada/Xt architecture in terms of system-independent package specifications, and to describe the analysis which contributed to major design decisions. The emphasis on system-independent package specifications rather than language independent specifications derives from recognition that the C language interfaces defined in the X Toolkit (Xt) Intrinsics definition are nearly sufficiently language independent -- for languages in the Algol tradition (including Ada). The Ada toolkit design verifies this claim, since there is a very close syntactic mapping of types and interfaces from the Ada specification to the C specification.

14. SUBJECT TERMS
X Window System
Ada/Xt Design
Widgets

15. NUMBER OF PAGES
87

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

PREFACE

This document was produced by Unisys Defense Systems in support of the STARS Prime contract under the Process Environment Integration task (UR20). This document "Ada Xt Architecture: Design Report", type A005 (Informal Technical Data) is provided as additional documentation for CDRL 01000, type A014 (Ada Package Specification/Source Code) which has is an electronic delivery to the STARS repository.

This document and source code were produced by Unisys Defense Systems at the Valley Forge Research facility in Paoli, PA and have been reviewed and approved by the following Unisys personnel:

Prepared by: Kurt Wallnau
Unisys Corporation

Reviewed by: Teri F. Payton, System Architect

Approved by: Billie Rhae Pruett for
Hans W. Polzer, Program Manager

TASK: UR20
CDRL: 01000

STARS-RC-01000/001/00

Ada/Xt Architecture: Design Report

for the

SOFTWARE TECHNOLOGY for ADAPTABLE, RELIABLE SYSTEMS (STARS)

Contract No. F19628-88-D-0031
Delivery Order 0002

Informal Technical Data (A005)

25 January 1990

Publication No. GR-7670-1107 (NP)

Prepared for:
Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:
Unisys Defense Systems
12010 Sunrise Valley Drive
Reston, VA 22091

TASK: UR20
CDRL: 01000

STARS-RC-01000/001/00

Ada/Xt Architecture: Design Report

for the

SOFTWARE TECHNOLOGY for ADAPTABLE, RELIABLE SYSTEMS (STARS)

Contract No. F19628-88-D-0031
Delivery Order 0002

Informal Technical Data (A005)

25 January 1990

Publication No. GR-7670-1107 (NP)

Prepared for:
Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:
Unisys Defense Systems
12010 Sunrise Valley Drive
Reston, VA 22091

Ada/Xt Architecture: Design Report
A005

UR20-User Interface Subtask

Unisys Defense Systems, E&ISG
Valley Forge Operations

Contents

1 Introduction 4

2 Overview of the X Window System 4
2.1 X and Industry Standardized API . . . 7
2.2 X Toolkits and the MIT X Toolkit "Xt" . . . 8
2.3 Why an Ada Implementation of Xt? . . . 9

3 Design Approach 10
3.1 Functional Specifications and Re-Engineering . . . 11
3.2 Object-Orientation in Ada/Xt . . . 12
3.3 Simulating Subprogram Types in Ada . . . 13
3.3.1 Task Types and Unchecked Conversion . . . 13
3.3.2 Ada Generics . . . 14
3.3.3 Cascading Generics and Case Statement Simulation . . . 17
3.3.4 System Dependent Programming . . . 20
3.4 Inheritance and Polymorphism in Ada/Xt . . . 22
3.4.1 Pseudo-Types and Compiled Data Structures . . . 23
3.4.2 Packaging Widget Type Definitions . . . 25
3.4.3 Subprogram and Resource Inheritance . . . 25

4 The Ada/Xt Design 27
4.1 Widgets . . . 32
4.1.1 Core Widgets . . . 32
4.1.2 Composite Widgets . . . 34
4.1.3 Constraint Widgets . . . 36
4.1.4 Widget Class and Superclass Look Up . . . 37
4.2 Widget Instantiation . . . 38
4.2.1 Toolkit Initialization . . . 38
4.2.2 Loading the Resource Database . . . 39
4.2.3 Parsing the Command Line . . . 39
4.2.4 Creating Widgets . . . 39
4.3 Composite Widget Management . . . 40
4.3.1 Procedure Types in Composite Widgets . . . 41
4.4 Shell Widgets . . . 41
4.5 Pop-Up Widgets . . . 55
4.6 Geometry Management . . . 57
4.7 Event Management . . . 59
4.8 Callbacks . . . 61

4.9 Resource Management 63
4.9.1 Interface to Resources 63
4.9.2 Representation of Resource Lists 65
4.9.3 Resource Management Package Specification 66
4.10 Translation Management 68
4.11 Utility Functions 70

5 Appendix A: Case-Statement Procedure Types **74**

6 Appendix B: System-Dependent Procedure Types **77**

7 Appendix C: Simple Widget Definition **81**

List of Figures

1 NIST User Interface Reference Model 6
2 Generic Widget Creation Specification 14
3 Generic Widget Creation Instantiation 15
4 Generic Superclass-Subclass Chaining 16
5 Generic Superclass-Subclass Instantiation 16
6 Procedure Types – Generics Generation 18
7 Procedure Types – Sample Usage 19
8 Procedure Types – System Dependent 21
9 Ada/Xt Widet Type Model 24
10 Widget Types – Ada/Xt Packaging Convention 26
11 UR20-UI Ada/Xt Packaging Implementation 31

1 Introduction

This report provides a detailed description of the Ada/Xt toolkit architecture. The purpose of this report is to describe the Ada/Xt architecture in terms of system-independent package specifications, and to describe the analysis which contributed to major design decisions. This report is part 1 of a two part Ada/Xt design description; part 2 consists of the compilable Ada package specifications of the UR-20 Ada/Xt implementation which conforms to the system-independent specifications outline in section 4 of this report.

The emphasis on *system*-independent package specifications rather than *language* independent specifications derives from recognition that the C language interfaces defined in the X Toolkit (Xt) Intrinsics definition [3] is nearly sufficiently language independent – for languages in the Algol tradition (including Ada). The Ada toolkit design verifies this claim, since there is a very close syntactic mapping of types and interfaces from the Ada specification to the C specification.

The difficult problems addressed by the UR20-UI design effort concerned the development of object-oriented toolkit features in Ada. This area of toolkit design exposed most of the language dependencies embedded in the C definition of Xt. A substantial part of this report discusses the effective use of Ada to provide object-oriented features (e.g., inheritance, procedure types) without unduely impacting toolkit performance and system independence, and without relying on automatic program generation techniques.

Section 2 of this report provides a high-level description of the MIT X Window System, including rationale for implementing a toolkit in Ada. Section 3 discusses the UR20-UI design *goals*, *approach*, and a *detailed analysis* of the key design decisions made concerning the implementation of object-oriented toolkit features. Section 4 provides the system-independent toolkit specification. The organization of section 4 parallels exactly the MIT X Toolkit Intrinsics documentation [3] Indeed, most operations defined in section 4 are not documented in this report; instead, the MIT documentation is referenced. This demonstration of toolkit similarity is an important result of the UR20-UI approach, and should facilitate MIT X Consortium acceptance of the STARS Ada/Xt toolkit for eventual Consortium maintenance and distribution.

2 Overview of the X Window System

This section of the report provides a brief description of the X Window System, and introduces and defines terminology used throughout this report. A more complete description of X can be found in [6].

X is a network-based windowing system. The National Institute for Standards and Technology (NIST) has developed a layered model to describe user interface architectures. This layered model is depicted in figure 1, and has become a federal information processing stan-

dard (FIPS) [4]. The lower four layers of this model in effect provide a description of the X Window System.

The lowest layer in the model, layer 0, is the X protocol. This data-stream protocol for X is currently undergoing formal standardization efforts in the ANSI X3H3.6 committee. The protocol defines the manner in which X *applications* communicate with X *servers*. Applications in this sense are sometimes called *clients*, although this terminology may be confusing because the term "client" is also used to describe user's of toolkit widgets (described, below).

The next layer, layer 1, is the programmatic interface to the protocol layer. This set of interfaces, known collectively as "Xlib", provides the primitive programmatic layer upon which X applications can make requests of X servers. SAIC, under a STARS Foundations contract, developed Ada bindings to the C Xlib implementation. This set of bindings can be characterized as (moderately) "deep" bindings, since a substantial effort was made to map the C data types to Ada, and do as much Xlib processing in Ada as possible before sending the actual request to the C implementation.

The Ada/Xlib bindings were not complete, however. Utility functions requiring procedure types as parameters were not implemented, probably because any implementation of procedure types that would enable C Xlib code to execute Ada subprograms was deemed to be too system-dependent for STARS. Resource management was also not implemented. The reason why these interfaces were not mapped in the Ada bindings is not clear, although it is possible that these interfaces were not officially part of the Xlib layer at the time the foundations work began.

The next two layers, layers 2 and 3, map to the MIT Toolkit *intrinsics* and *widgets* layers, respectively. The Unisys UR20 user interface subtask addresses the development of an Ada *implementation* of these two layers – not bindings. These Ada implementations will make use of the SAIC Xlib bindings, which have been upgraded to revision 3 (X11R3), and extended to include all of the Xlib resource management functions, as part of the UR20 task.

Layer 2, the subroutine foundation layer, defines a single application portability interface (API) for manipulating collections of user interface abstractions (called "toolkit" in the reference model, but called "widgets" or "widget sets" in Xt parlance). This separation of subroutine interface from user interface abstraction presents interesting programming paradigm questions to Ada software developers. That is, the intrinsics (layer 2) define a set of interfaces for manipulating an open-ended set of abstractions. As will be discussed later in this report, Ada generics are not sufficiently general to support this kind of software layering.

Layer 3, the toolkit, is essentially a library of reusable, extensible, and composable user interface abstractions. This "widget" library provides an excellent example of what Unisys has been describing as *tool fragments*, here tailored to the fragments of tools which concern the display of application program data. The ideas of widget reusability, extensibility and composability are central to the MIT X Toolkit, or "Xt", which draws heavily upon the concepts of object-oriented programming languages and systems in order to achieve these

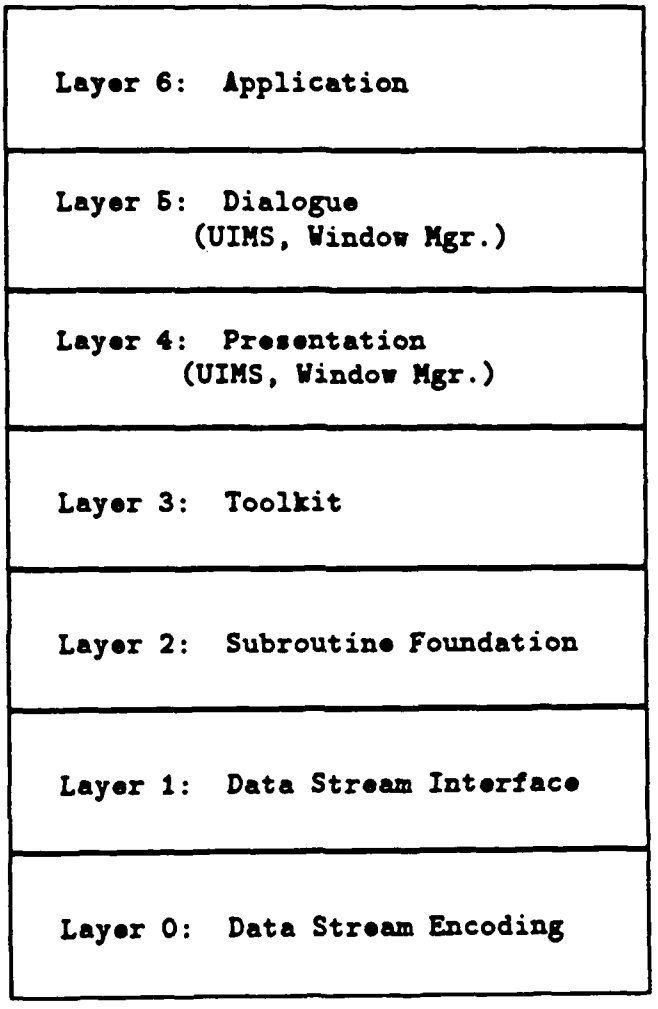


Figure 1: NIST User Interface Reference Model

effects. A significant part of this design report addresses the simulation of object-orientation in Ada without recourse to program generation.

2.1 X and Industry Standardized API

In recent years the industry has recognized the importance of separating the user interface code and the application code. This separation reduces software development costs by permitting reuse of user interface code and providing consistent behavior (reducing training costs and reducing errors caused by differences in the user interface behavior). This separation requires a well defined interface between the application and user interface abstractions, called an application program interface or API.

In X the interface to the user interface abstractions or widgets consists of functions (contained in the intrinsics) to access widgets and the data structures within the widgets that allow applications to customize the user interface. These data structures contain resources (specifying color, size, fonts, etc.) and lists of application procedures invoked by the intrinsics upon occurrance of specified events.

Why is the industry demanding a standard API and user interface abstractions? The demand originates in the user community (especially the U. S. government) and independent software vendors (ISVs). Users now buy hardware platforms and software from a variety of vendors and need a consistent user interface ("look and feel" or appearance and behavior) across platforms and among applications running on a single platform.

ISVs face a demand for their applications on a variety of platforms. The application changes little from platform to platform, but the user interface may change drastically. The separation of the user interface from the application is important, so that a port to another platform requires at most a rewrite of the user interface and not a complete rewrite of the application. Furthermore, a common API means the application code need not change when the user interface changes. Portable user interface abstractions like the Xt widgets mean that an ISV can easily port a user interface from platform to platform, thus reducing the conversion costs and ensuring a consistent "look and feel" across all platforms, and consistency among the ISV's product user interfaces.

Industry standards organizations, ANSI X3 and IEEE, are standardizing the lower layers of the FIPS model. The X protocol (data stream encoding) standards work began two years ago and should be completed soon by X3H3.6. The toolkit and API standardization work began in 1989 in the IEEE P1201 committee. P1201 is currently working on a standard for the widget set. ANSI X3V1 is working on standards for man-machine interfaces. X3V1.9 will develop standards for "look and feel" of user interface abstractions such as menus.

Why has NIST and the standards bodies chosen X? X is the first widely accepted window system addressing the needs of the networked, bit-mapped graphics workstation environment. X runs on a wide range of Unix based platforms, Digital VMS and Ultrix machines, IBM mainframes, Apple MacIntosh, PCs and graphics terminals. One reason for its acceptance is

that source is available free of licenses or royalties. In addition, interfaces to several languages exist, C (primarily), C++, Lisp, Ada, and Prolog; however not all implementations support the Xt toolkit layer.

2.2 X Toolkits and the MIT X Toolkit "Xt"

A number of toolkits evolved from the X Window System. Most of these are based on the Xt intrinsics. MIT released a sample set of widgets, Athena widgets, with the Xt Intrinsics, and many applications were and still are being written using Athena widgets. The Athena widgets were an incomplete set (there is no menu widget; one can be built from other Athena widgets), and so various companies added widgets to their X based products. Digital developed XUI, AT&T Xt+, and HP and Sony developed widget sets. In many cases (XUI and Xt+) the intrinsics were extended. With the demand for a single API and "look and feel" from users and ISVs, groups like the Open Software Foundation (OSF) moved toward a single API and widget set. OSF developed Motif by merging XUI and the HP widget set. There is still not an agreement on a single widget set, but the standards work will eventually define one.

Most toolkits are based on the MIT Xt Intrinsics, but several are not: Xray [2], Andrew [5] and XView are well known examples. Xray (or Xrlib) was an early HP toolkit which added three layers above the Xlib layer:

- Intrinsics - input handling, object interaction and geometry management,
- Field Editors - the basic user interface abstractions such as scrollbars and buttons,
- Dialogs - higher level abstractions such as menus, message boxes and panels.

With the rising popularity of Xt, HP implemented a similar "look and feel" with an Xt Intrinsics based widget set and use of Xray has declined.

Sun recently announced the release of XView, a toolkit built upon the Xlib layer. The XView API is compatible with the proprietary SunView API. XView implements the Sun/AT&T Open Look "look and feel".

The Andrew Toolkit, developed at Carnegie Mellon University (CMU), is a window system independent, object oriented toolkit. Besides a CMU built window system, the Andrew Toolkit supports X (X protocol and Xlib). The Andrew Toolkit consists of *data object/view* pairs where the *data object* is the information to be displayed (text in a text editor, for example) and a *view* is the user interface abstraction (scrollbars, menus, etc). One feature of Andrew is the ability to intersperse multiple *data objects* within a *view*. This permits a mixture of text, graphics and animations within a window (e.g. animation and graphics can occur along with text in the body of an email message). This toolkit has not been widely accepted beyond CMU.

Xt is clearly the dominant toolkit in the marketplace. What are the features that are making Xt the *de facto* standard and soon an official standard? Certainly, its availability free of licenses and royalties is a contributing factor. Other important attractions are its extensibility through creation of new widgets, and from its object oriented design and the ability to easily subclass widgets. Adding new widgets or subclassing does not require recompilation of the toolkit since the intrinsics do not need to know anything about a specific widget. The intrinsics are policy free (implies nothing about "look and feel") so that a vendor or application is free to specify its own "look and feel" by choice of widgets. Xt also provides the separation of user interface objects from the application code, thus permitting portability and reuse of the user interface code.

2.3 Why an Ada Implementation of Xt?

Implementing Xt in Ada presents some challenging problems, and is not without some risk. To achieve a high degree of flexibility and extensibility, Xt made use of language features which have only tenuous analogs in Ada (e.g., procedure types). Finding the correct Ada approach to these language dependent features of Xt requires making tradeoffs among: compiler independence, operating system independence, and system/hardware independence. Choosing a suboptimal Ada design/implementation risks industry acceptance of Ada/Xt, and hence risks emergence of a *de facto* Ada toolkit API. However, in many cases, no "perfect" solution exists. In fact, a significant portion of this report deals with the tradeoffs among various competing implementation strategies.

Since a fairly substantial engineering effort must be expended (with some risk) to implement Xt in Ada, a reasonable question to ask is whether a better cost/benefit ratio would be obtained by following the Ada/Xlib example and developing an Ada binding to the MIT X toolkit. There are several reasons why a bindings approach to Xt would not be adequate in the long-term.

First, there is the issue of widget set extensibility. A significant feature of the Xt model is the ease with which new widgets can be constructed from old widgets. Indeed, this is a hallmark of object-oriented programming in general, which attempts to maximize reuse by factoring abstractions into class hierarchies, facilitating finer-grained reuse than possible with unstructured collections of monolithic abstractions. However, a set of bindings to the C implementation of Xt would require that new widgets be programmed in C, and that a fairly elaborate system-dependent type mapping interface be developed and maintained which maps Ada application resource types to the underlying C representations for management by the C toolkit implementation.

Second, beside issues of static inter-language interfaces, there are issues of inter-language *runtime* cooperation. The notion of procedure reference is indelible in Xt: a major part of the toolkit model is that the toolkit will execute application code *on behalf* of the application *in response* to certain events generated by the server. That is, an Xt application program

does not have a traditional control structure, but rather specifies what code to execute under various circumstances: the toolkit in effect executes the application program. As a consequence, a toolkit binding needs to provide the C implementation with the *address* of a subprogram which will execute Ada code (either directly or indirectly). This introduces a considerable degree of compiler dependence, since the details of parameter passing protocol and stack frame environments are not specified by the Ada language definition. Although the UR20 toolkit implementation makes use of procedure addresses, it provides an interface which will accommodate a "pure Ada" solution should circumstances demand it. An Ada binding to Xt would not have this luxury.

Finally, there are issues of runtime environment interaction. If a significant number of Ada applications are to be developed which use the X Window System, issues of conflict between the Ada runtime system and underlying host operating system must be considered. For example, conflicts between the Ada runtime environment usage of Unix signals to control task scheduling and C Xlib code must be carefully considered. Some Ada runtime environments even provide a mechanism to disable runtime environment signal usage during calls to external language routines. This added complexity and system dependency can only be adequately resolved if, in the long term, a full Ada implementation of the client interfaces to X is developed (Xlib and toolkit interfaces). From this perspective, the development of another Ada binding would be at most a stopgap effort which would have to be redone at a later date.

3 Design Approach

The UR20 task had two overriding goals to achieve in the Ada toolkit implementation:

- Develop an eminently usable implementation of Xt which fully preserves the features and advantages of the C implementation.
- Establish that the Ada implementation faithfully implements the MIT toolkit, and achieve X Consortium acceptance of the Ada implementation as a Consortium "product."

The first goal does not strictly require implementing Xt, but rather some toolkit which provides all of the features that Xt provides. In practice however, it would be unreasonable to expect to design and implement an entirely new toolkit model (in a nine month performance period) which can compete with Xt. Instead, this goal had more to do with ensuring that the resulting implementation of Xt sufficiently preserves the strictures of Ada style and usage, and was also efficient and compact so as not to overburden applications which require interactive windowing interfaces. In short, this goal concerns acceptance of Ada/Xt by Ada software practitioners.

The second goal requires presentation to the X Consortium of some evidence that the Ada implementation faithfully implements the MIT X Toolkit. This presented an interesting question, since the only existing description of what Xt *is* is a very detailed description of the C implementation. That is, there is no pre-existing specification of what Xt is beyond it's implementation. This goal, then, concerns provision of an architectural description of the Ada implementation which will maximize our chances of attaining acceptance of Ada/Xt by X Consortium members.

Note: This section of the design report provides an in-depth discussion of various approaches to implement Xt in Ada. In places, understanding the highly detailed discussion of advanced Ada programming techniques used in Ada/Xt requires that the reader has a significant degree of Ada and MIT X Toolkit competency.

3.1 Functional Specifications and Re-Engineering

The originally stated UR20-UI toolkit design approach was to begin with a study of the MIT Xt implementation and documentation, and from this study extract and specify a *language independent specification* of the MIT X Toolkit. This specification would then be mapped to some Ada realization. Thus, two products of this design process were envisioned: a language independent specification, and an Ada language specification, of Xt.

However, this abstract process encountered difficulties early on. It became apparent that there was very little that was indeed "language independent" in the C documentation. Further, we began to suspect that what truly did constitute language independent architectural constraints would, when specified, provide little or no insight to the task of creating an Ada implementation. Thus, our premise that a language independent model could be extracted via a reengineering process seems faulty.

For example, consider the issue of widget subclassing and inheritance. The C implementation supports subclassing and inheritance via manual type conversions to predefined, known widget and widget-class data structures, and data structure specification conventions, respectively. This is a concrete realization of the abstract idea of widget-class hierarchies with inheritance. The realization of this abstract architectural feature in an object-oriented language like C++ or CLOS will likely be radically different from the C realization; a language-independent specification sufficiently general to describe these various implementations would be vague to the point of being useless as a prescriptive vehicle.

This may appear to be a disappointing result, but there are positive aspects. First, we concluded that a truly language independent specification is not likely to be of much use beyond shallow conformance testing. However, the Xt implementation does provide an approach for implementing an object-oriented system in non-object-oriented languages. More specifically, we were able to test the assertion made by the author's of Xt that the intrinsics are language independent for procedural languages.

We conclude that they are indeed *reasonably* language independent for procedural languages in the Algol-Pascal-Modula2-Ada tradition. Further, modulo minor syntactic variations (e.g., turning C functions with side-effects into Ada procedures), an Ada implementation of Xt is able to preserve a very direct syntactic mapping of Ada intrinsics to C intrinsics, and Ada widget programming conventions to C widget programming conventions. This is indeed an excellent result because it provides in effect what we wanted from the language-independent specification: some means to justify to the X Consortium that we had implemented the MIT X Toolkit, and not some new variant. Further, this close correspondence is a compelling argument for tying evolution of the Ada implementation to it's "parent" C implementation, since the differences are not so great as to make parallel evolution an unreasonably expensive venture.

Although UR20-UI does not provide a language independent specification for Xt, we do provide a system-independent Ada specification, suitable as a basis for standardization in the Ada community. This specification is useful for highlighting where extra implementation details may be added to support implementation on a particular hardware/operating system platform.

3.2 Object-Orientation in Ada/Xt

It is not the purpose of this design report to convince the reader of the utility of object-oriented programming in the development of user interface-intensive systems. This report also assumes some level of familiarity with such terms as object *class*, object *instance*, and *inheritance*. Description of object-oriented concepts are numerous in literature, of which [1, 7] are just a small (but significant) portion.

Three features of object-oriented languages need to be *simulated* in Ada before an adequate implementation of Xt can be undertaken. The design of this simulation using Ada features, rather than through some form of program generation, constituted a significant portion of the Ada/Xt design process. These three features are:

- subprogram *types*, i.e., "methods"
- inheritance
- polymorphism

It is important to note the term *simulation*. The ideas of runtime type polymorphism and type inheritance introduces a model of type semantics not implemented by Ada (or C). Since Ada does not implement the type model required of an object oriented system, this type model must be simulated. Thus, one way of viewing the toolkit architecture is as a system of programmatic interfaces and programming conventions to use these interfaces in order to simulate object-oriented capabilities in a non-objective language.

The remainder of this section describes how the Ada/Xt architecture simulates subprogram types, inheritance, and polymorphism.

3.3 Simulating Subprogram Types in Ada

This section describes various solutions to the problem of simulating subprogram types in Ada. It is important to note that the problem being solved is not simply that of referring to executable code as data. Were that the case, the Ada tasking mechanism would be sufficient (albeit somewhat an overkill). Instead, the problem is one of referring to subprograms as types characterized by interface alone, such that two subprograms with the same interface but that compute distinct functions would be considered subprograms of the same *type*.

3.3.1 Task Types and Unchecked Conversion

One implementation approach makes use of Ada task types as a foundation for implementing subprogram types. Unfortunately, task types do not provide for alternative task bodies for task specifications.

One way around this is to define a task type *T0* with a task entry *Te* whose parameter profile matches the subprogram profile type being implemented. Rendezvous with instances of *T0* on entry *Te* will raise an exception – task type T0 is merely used in order to create a type mark for constructing a data structure containing references to other task types which share the same *syntactic* task specification. New task types *Tn, Tm, Tp* can be defined which have the same syntactic specification as T0, and instances of these new task types can be type converted (via Ada unchecked_conversion) to instances of type T0. Finally, the entries of Tn, Tm, and Tp are accessed via rendezvous with these task instances *as if* they were instances of T0. Thus, unchecked_conversion is used to achieve distinct task bodies for the same task specification.

However, this implementation was rejected for two reasons. First, although this technique worked on several compilers, this use of unchecked conversion is clearly outside the scope of the *intended use* of this feature. There is no reason to believe that all compiler vendors will generate code to perform rendezvous based solely upon the syntactic form of the task definition. Although we do endorse some level of system-dependent programming for Ada implementations of the X toolkit, this kind of systems-dependent programming must be considered dangerous.

A second reason for disqualifying this technique derives from practical constraints imposed upon the UR20 approach. That is, UR20 takes as a foundational basis the STARS Foundations Ada/Xlib bindings. Thus a significant amount of application processing is actually done by code written in C. However, the use of tasking introduces many potential conflicts between the Ada runtime environment and the underlying host operating system, which is accessed directly by the C implementation of Xlib. In particular, Ada runtime en-

```
generic
    type gen_widget is private;
    type gen_widget_class is private;

    -- core class procedures needed during widget creation
    with procedure class_part_initialize(wc : widget_class);
    with procedure class_initialize;
    with procedure initialize(request, new_request : widget);
    with procedure initialize_hook(w : widget;
                                   args : arg_list);
    -- creates the widget record with proper size
    with function malloc_widget return gen_widget;
package create is
    function xtcreatewidget(name : string;
                            widget_class_ptr : gen_widget_class;
                            parent : widget;
                            args : arg_list) return widget;
end create;
```

Figure 2: Generic Widget Creation Specification

vironment use of Unix signals as a means of task scheduling (e.g., SIGALARM) can conflict with the smooth execution of C code depending upon Unix interprocess communication.

3.3.2 Ada Generics

Another approach to handling procedure types is through the use of Ada generics to parameterize widget class definitions with the class operations that would be otherwise represented as procedure type instances embedded in the widget class data structures (these data structures are described in greater detail later in this report). Although this static parameterization would not apply to more dynamic uses of procedure types (e.g., callback resources), generic parameterization of static subprogram types would constitute a significant design decision for the Ada/Xt toolkit. We tried this with the procedure types in the *core class* record structure. These procedures handle initializations, setting and retrieving resource values, resizing, exposures, etc. We thought certain functional areas, such as widget creation, could be defined by generics and instantiated with the needed functions defined in the *core class*. To minimize the size of the generics we attempted to write the generics as a thin generic interface which references underlying, non-generic widget creation code.

The code fragment in figure 2 is the generic specification for widget creation code. The code fragment in figure 3 is an instantiation of the generic specification for label widget creation.

Some of the *core class* procedure types were only invoked within superclass to subclass chains and could be implemented as generics parameterized by the *core class* function and

```
package label is
    package label_create is new create(label_widget,
                                       label_widget_class,
                                       class_part_init,
                                       class_initialize,
                                       init,
                                       init_hook,
                                       malloc_label_widget);
end label;
```

Figure 3: Generic Widget Creation Instantiation

its superclass instantiation of the same chaining generic. The code fragment in figure 4 is the superclass to subclass chaining generic for the *class_part_initialize* function specified in the *core class* of every widget class.

Finally, The code fragment in figure 5 instantiates the *class_part_init* superclass-to-subclass chain for the label widget. Note, the label widget does not execute any code for *class part* initialization and a dummy procedure (with empty body) is used to instantiate the generic.

Although we demonstrated the use of generics for procedure types in the proof-of-concept for widget creation, we exposed a number of inadequacies in the approach.

We realized that this approach would require a large number of generic instantiations for each widget class used in an application. We hoped that the generics would be a thin interface to the intrinsics, but the widget creation generic showed that references to generic parameters were needed throughout the widget creation code. We concluded that generic Ada packages to simulate procedure types require virtually complete implementation of the intrinsics within generics, thus forcing applications to instantiate a copy of the intrinsics for each widget class used in the application.

Simulating procedure types with generics failed to handle all uses of procedure types in the intrinsics. Our method worked because the procedures were determined at compile time and based on a static tree structure (the *widget class* hierarchy). This failed on dynamic structures such as the run time widget tree. The generic approach failed in widget creation when calling the widget's parent's *insert_child* procedure. The generic instantiation of *XtCreateWidget* can not know anything about the parent's *insert_child* procedure. Procedure types referenced via dynamic structures, such as the widget tree, require a different approach.

The expected code size due to the large number and size of generic instantiations and the failure of the generic approach for simulating some procedure types made this approach unacceptable.

```
package intrinsics is
    -- superclass to subclass chaining generics
    generic
        with procedure superclass_class_part_init(class : widget_class);
        with procedure class_part_initialize(class : widget_class);
      procedure class_part_init_chain(class : widget_class);
end intrinsics;

package body intrinsics is
   procedure class_part_init_chain(class : widget_class) is
   -- class_part_initialize is a downward chaining procedure
       class_ptr : core_class := widget_to_core_class(class);
   begin
       if class_ptr.core_class.superclass /= null_address then
          superclass_class_part_init(class_ptr.core_class.superclass);
       end if;
       if class_ptr.core_class.class_part_initialize  then
          class_part_initialize(class);
       end if;
   end class_part_init_chain;
end intrinsics;
```

Figure 4: Generic Superclass-Subclass Chaining

```
package label is
    label_core_part : core_class_part :=
          (superclass         => superclass_to_widget_class(simple_classrec_ptr),
           class_name         => "label",
           widget_size        => labelrec'size,
           class_initialize   => true,
           class_part_initialize => false,
            -- remaining fields follow
           );
    -- downward chaining functions
    -- renames the label widget's superclass class_part_init function
    procedure superclass_class_part_init(class : widget_class)
                                    renames simple.class_part_init;
    -- the label widget's class_part_init function is null so instantiate
    -- the generic with the superclass's and a dummy label class_part_init
    procedure class_part_init is
              new class_part_init_chain(superclass_class_part_init,
                                        null_class_part_initialize);
end label;
```

Figure 5: Generic Superclass-Subclass Instantiation

3.3.3 Cascading Generics and Case Statement Simulation

Although Ada does not provide for procedure types (from which reference types can be constructed), it is still possible to simulate procedure types in a system-independent manner. The simple scheme is to assign a unique identifier to subprograms, and use this identifier as an index to an Ada case statement which invokes the subprograms. The only difficulty that needs to be addressed in this implementation is *how* these unique identifiers are generated.

In a Q-task standards report [8] Unisys proposed one implementation which uses the Ada generics mechanism to generate procedure references. The package specification for this implementation is provided in figure 6. The full implementation is provided for convenience in appendix A.

This implementation makes use of *cascaded* generic instantiations to in effect create a linked-list of generated (via instantations) package bodies. Each generated package body acts as a state machine which manages a discrete range of subprogram indexes; indexes that lie outside this range indicate that the actual subprogram referenced by the index is managed by a different instantiation, which is then accessed via a "next_callback" operation provided as a generic actual from a previous (cascaded) instantiation.

Since this explanation may be obscure, an example usage of this implementation is provided in figure 7. Note that this usage generates only one procedure reference per instantiation, despite the fact that the generic interface allows as many as three procedure references to be generated. This is done for simplicity to illustrate the use of cascaded generics as a means of achieving an open-ended mechanism for attaining procedure references.

One advantage of this approach (beyond it's pure use of Ada) is that application programmers (i.e., toolkit clients) can add application-defined subprogram type instances to cascades of pre-defined (i.e., by the toolkit intrinsics or widget programmers) subprogram type instances. This reduces potential configuration management problems that would be introduced if all subprogram type instances needed to share the same case statement dispatcher.

One problem with this implementation concerns the visibility of the top-level (i.e., "last") generic instantiation in the cascade. For systems such as Xt, which execute client subprograms based upon event sequences generated from the server, client defined subprogram references must be visible to the toolkit intrinsics, in addition to widget and intrinsics defined subprogram references. As a result, the toolkit must either be (at some level) a generic abstraction which parameterizes the "call" operation for each procedure type, or else the application programmer must complete the implementation of a top-level non-generic call interface whose implementation will reference the top-level generic cascade.

Turning the intrinsics into a generic abstraction is not tenable for reasons which were discussed in the previous section. Requiring the application programmer to complete the procedure call implementations appears to introduce a considerable degree of inconvenience, but perhaps this is outweighed by the added level of machine independence. The UR-20

```
package callback_mechanism is

   CALLBACK_CALL_ERROR: exception;
   CALLBACK_INSTALL_ERROR: exception;
   CALLBACK_RANGE_ERROR: exception;

   MAX_CALLBACKS: constant:= 1024;
   NUM_CALLBACKS: constant:= 3;

   subtype callback_id_range is natural range 0 .. MAX_CALLBACKS;

   package callback_ids is
      type callback_id_type is private;
      null_id: constant callback_id_type;

      function to_callback_id_range(id: callback_id_type)
         return callback_id_range;

      private
         function next_callback_id return callback_id_range;
         type callback_id_type is record
            the_callback_id: callback_id_range:= next_callback_id;
         end record;
         null_id: constant callback_id_type:=
            (the_callback_id => callback_id_range'first);
   end callback_ids;

   use callback_ids;

   -- the default procedures will never actually be called
   procedure default_next_call_back(id: callback_id_type; s: string);
   procedure default_callback(s: string);

   generic
      with procedure cb1(s: string) is default_callback;
      id1 : in callback_id_type:= null_id;
      with procedure cb2(s: string) is default_callback;
      id2 : in callback_id_type:= null_id;
      with procedure cb3(s: string) is default_callback;
      id3 : in callback_id_type:= null_id;
      with procedure next_callback(id: callback_id_type; s: string)
         is default_next_call_back;

      package callbacks is
         procedure callback (id : callback_id_type; s: string);
      end callbacks;

end callback_mechanism;
```

Figure 6: Procedure Types – Generics Generation

```
with callback_mechanism; use callback_mechanism;
with text_io; use text_io;
procedure test_callback_mechanism is
   use callback_ids;

   procedure p(s: string);
   procedure q(s: string);

   p1: callback_id_type; -- p1 and q1 now have valid callback ids
   q1: callback_id_type;

   -- in p_callbacks, cb2 and cb3 are "default" callbacks
   package p_callbacks is new callbacks(cb1 => p, id1 => p1);

   -- q_callbacks uses p_callbacks callback routine to chain instantiations
   -- procedure p and q could have both been installed in a single
   -- instantiation, but we're demonstrating instantiation chaining.
   package q_callbacks is new callbacks(
      cb1 => q,
      id1 => q1,
      next_callback => p_callbacks.callback);

   use q_callbacks; -- make the last instantiation directly visible

   -- procedures p and q do different things
   procedure p(s: string) is
   begin
      put_line("P:" & s);
   end p;
   procedure q(s: string) is
   begin
      put_line("Q:" & s);
   end q;

begin
   callback(p1, "hello world");
   callback(q1, "hello world");
end test_callback_mechanism;
```

Figure 7: Procedure Types – Sample Usage

toolkit does not use the cascading generics implementation; however, the procedure invocation interface actually used is not inconsistent with the cascading generics implementation.

Another apparent problem concerns performance. A considerable amount of code needs to be executed just to locate the appropriate procedure to execute. This problem is compounded if several cascades of generics are needed. Although the generic abstraction can be implemented to accommodate more procedure formals, architectural considerations may require several cascades, e.g., one instantiation for intrinsics defined subprogram instances, one for widget defined instances (perhaps one for each widget type), and finally at least one for the application itself

3.3.4 System Dependent Programming

Yet another alternative implementation scheme for introducing subprogram types takes advantage of the system-dependent Ada attribute *'address*, applied to subprogram instances. Although this attribute "refers to the machine code associated with the corresponding body," (Ada LRM), this definition leaves open considerable compiler implementation leeway, and so any solution based upon this feature is inherently non-portable, both across compilation systems and host environments.

Nevertheless, this form of system dependent programming appears to be more justifiable than, for example, the unchecked programming used in the task-based simulation noted above. Also, although non-portable, the amount of system-dependent code required to implement subprogram types with the *'address* attribute appears to be rather small, assuming the compilation system provides adequate documentation on procedure call conventions (and does indeed implement the *'address* attribute in a *reasonable* way).

The package specification for a sample subprogram type is provided in figure 8. (This subprogram type will be referred to later in this report when runtime inheritance is described). The package body is provided in appendix B for the VADS, TeleSoft, and Tartan compilers. [1] The procedure control block structure is tailored for use with Alsys, although the same data structure also works for VADS and TeleSoft (the data fields are not used in these compilers).

The interpretation of this implementation is very similar to that described in the cascaded generics implementation: the generic instantiation generates a unique identifier for the subprogram type instance. In fact, the package interface is nearly identical, and for practical purposes the implementations are interchangable (which is convenient, in case the system-dependent approach is unworkable for a given compiler). The major difference is that the implementation of the "call" subprogram dispatch procedure will be in C or assembly language, or some language which can de-reference subprogram addresses (in effect, execute a "jump subroutine" or "jsr" instruction).

[1] The Alsys implementation is more convoluted due to difficulties in getting documentation on the Alsys procedure call conventions.

```
package xt_procedure_types is

   -- vendor-specific procedure control block:
   type procedure_control_block is record
      proc_address    : system.address;
      -- subprogram environment context data fields here...
   end record;

   package xt_widget_class_procs is

      type xt_widget_class_proc_rep is limited private;
      type xt_widget_class_proc is access xt_widget_class_proc_rep;

      -- A constant used for runtime inheritance resolution:
      function xt_inherit_widget_class_proc return xt_widget_class_proc;

      -- the subprogram dispatch function:
      procedure call(the_proc_id      : xt_widget_class_proc;
                     the_widget_class : widget_class);

      -- this generic is instantiated with the procedure to be called
      -- proc_id will acquire the address of the the_proc as well as
      -- the (activation frame) environment needed to execute the_proc
      generic
         proc_id : in out xt_widget_class_proc;
         with procedure the_proc(the_widget_class : widget_class);
      package procedure_pointer is
      end procedure_pointer;

   private

      type xt_widget_class_proc_rep is new procedure_control_block;

   end xt_widget_class_procs;

   -- other procedure type definitions follow...

end xt_procedure_types;
```

Figure 8: Procedure Types – System Dependent

It should not be concluded that this mechanism is completely without restrictions. In fact, several very subtle problems can be encountered, and great care must be taken in defining procedure types for use with this implementation.

One major area of concern is related to parameter passing conventions: successful use of this implementation *requires* an adequately documented compilation system. For example, an unconstrained array can be implemented as an array with a dope vector; such arrays can quite naturally be passed to subprograms via *two* parameters, not one. Thus, in general the foreign language "call" routine will have to be tailored for each subprogram type to accommodate passing different numbers and types of arguments.

Note: The UR20 implementation actually employs a single dispatch function that invokes ada subprograms via their address, and passes only a single argument – the address of a record structure which encapsulates the set of arguments defined for various procedure types. See appendix B for the details of this optimization.

A second area of concern is related to the consequences of Ada subprogram and type elaboration issues. Essentially, safe use of this implementation requires the programmer ensure that the subprogram body and all types used by the subprogram be fully elaborated before the subprogram reference is obtained via instantiation.

Finally, and perhaps most importantly, safe use of the system-dependent implementation requires thoughtful application of usage guidelines which ensure that the subprogram *object* will exist only in scopes compatible with the referenced subprogram. That is, care must be taken to ensure that the environment (e.g., stack activation frames) appropriate for the referenced subprogram exists at the time the subprogram is called. This could have been enforced in the language by providing the *call operation* as part of the generic subprogram type abstraction, rather than in a global context; however, this would have resulted in visibility problems described in the previous section on cascading generics. A simple usage guideline for the system-dependent implementation is to introduce the subprogram *object* declaration within the same scope which contains the subprogram declaration.

3.4 Inheritance and Polymorphism in Ada/Xt

The notions of inheritance and polymorphism are closely related in object oriented systems. Crudely, polymorphism describes a type model in which operations (in general, "*properties*") defined on a single type are applicable to a family of related types. Ada generics provide a form of *parametric* polymorphism, i.e., a set of operations is applicable to a type if that type is parameterized by a set of known properties, e.g., is assignable, has equality defined, has an ordering relation defined, etc. This is sufficient only if the types in a system can be characterized by a finite number of known properties. This is not the case in an object-oriented class hierarchy, which defines (via inheritance) an *inclusion* polymorphism type model.

The Ada/Xt toolkit simulates polymorphism and inheritance by means of a visible

pseudo-type hierarchy which provides a logical type system to support a (apparent) polymorphic external programmatic interface, and a private *actual-type* hierarchy to support the implementation of inheritance. That is, the pseudo-types are "public", i.e., visible in the API, whereas the actual types are "private", i.e., visible only to widget programmers. This usage of the terms public and private is analogous to Ada public and *private* types, but is not implemented in terms of Ada *private* types.

3.4.1 Pseudo-Types and Compiled Data Structures

The Ada/Xt type hierarchy is actually composed of four parallel hierarchies: a hierarchy of widget *class* pseudo-types, a hierarchy of widget *instance* pseudo types, and a hierarchy of actual widget class and instance types. The programmatic interfaces to the toolkit are implemented in terms of the pseudo-types, so called because in reality these types are merely placeholders for the actual types, defined by widget programmers.

Figure 9 provides a pictorial representation of the Ada toolkit type implementation. Widget class pseudo-types are arranged in an Ada *subtype* hierarchy; widget instance pseudo-types are arranged in a (parallel) *derived type* hierarchy. Within the implementation of the intrinsics and widgets, toolkit code performs unchecked conversions from Ada objects of type *pseudo-type* to the corresponding *actual* type in the parallel *actual-type* hierarchy.

This set of parallel type hierarchies provides a number of important features for the Ada/Xt implementation. The pseudo-type hierarchy allows the toolkit to be extensible with respect to widget sets. By defining the intrinsics operations in terms of pseudo-types rather than actual widget types (i.e., their Ada record representations), one set of intrinsics functions can manipulate an open-ended number of actual widget types.

A further refinement of the pseudo-type hierarchy is possible due to the Ada *derived type* feature. Occasionally, widget programmers may wish to define operations on widgets which are accessible directly to the application programmer. For example, text edit widgets may provide operations to retrieve the currently selected segment of text within the editor view. [2] In Ada/Xt, the widget programmer can define such operations on a widget. Subclasses of this widget will "inherit" operations via Ada type derivation on the pseudo-types. The operation will still be applicable on subclass instances because the code "inherited" will be performing an unchecked type conversion to a record structure which is layout-compatible with the subclass instance's actual type. Thus, the pseudo/actual-type hierarchy in effect augments Ada type derivation with type representation changes.

Note that the use of unchecked conversion from pseudo types to record type definitions in the parallel actual-type hierarchy requires strict widget programmer control over the layout

[2] Note: for the most part, application programmers make application operations available to the widgets to be executed by the widget or toolkit on behalf of the application. This kind of inversion is typical of event-based programming, and distinguishes toolkit programming from "traditional" procedure-based programming styles.

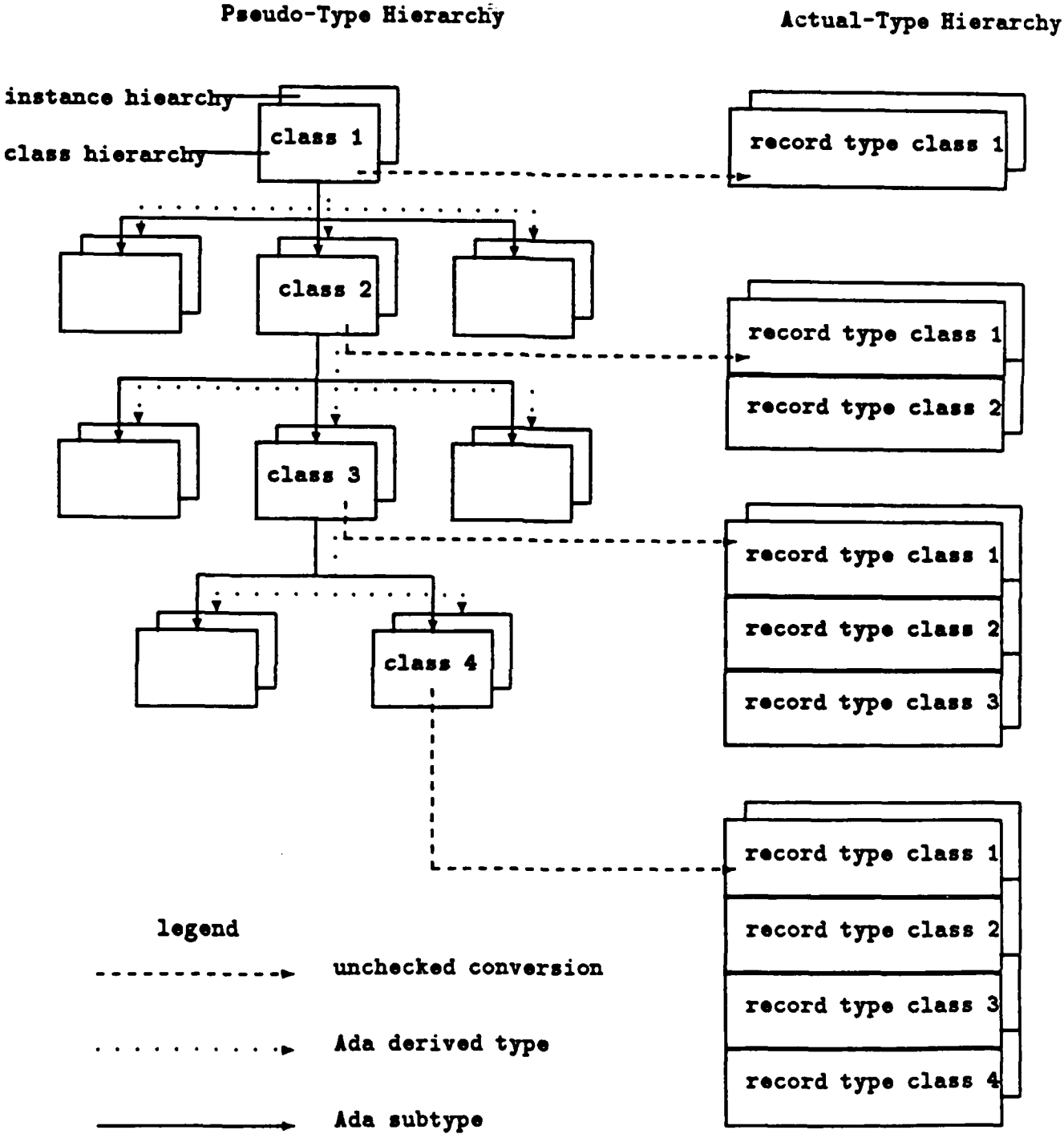


Figure 9: Ada/Xt Widet Type Model

of the actual-type record definitions. That is, for an operation defined on object class 1 (in figure 9) to work on an instance of object class 3, the actual representation of the record for instances of class 1 and class 3 must be identical for the common prefix fields, i.e., for the components defined in the record definition for class 1. Proper enforcement of this constraint only can be ensured through use of Ada record representation clauses.

3.4.2 Packaging Widget Type Definitions

As indicated, a widget type definition consists of four distinct hierarchies, arranged as a "public" and "private" widget definition interface. The public interface is defined in terms of the pseudo types, and defines operations that are to be available to the application programmer.

The private interface is defined in terms of Ada record definitions which characterize the object state for widget classes and instances. In order to ensure that widget subclasses share a common data structure prefix with their ancestors in the superclass-subclass hierarchy, the widget programmer must *explicitly* insert the record type definitions for the type hierarchy of the subclass's ancestors, and use Ada record representation clauses to ensure the relative ordering of these fields within the newly defined widget.

Figure 10 illustrates the packaging structure for defining widgets in the Ada/Xt toolkit. This mechanism is quite similar to the method used in the C implementation, with the major differences being the use of representation clauses, and the use of Ada derived types to automate some of the inheritance process. Note that the Ada "with" hierarchy parallels the *superclass-subclass* taxonomy defined by the logical widget class structure. [3]

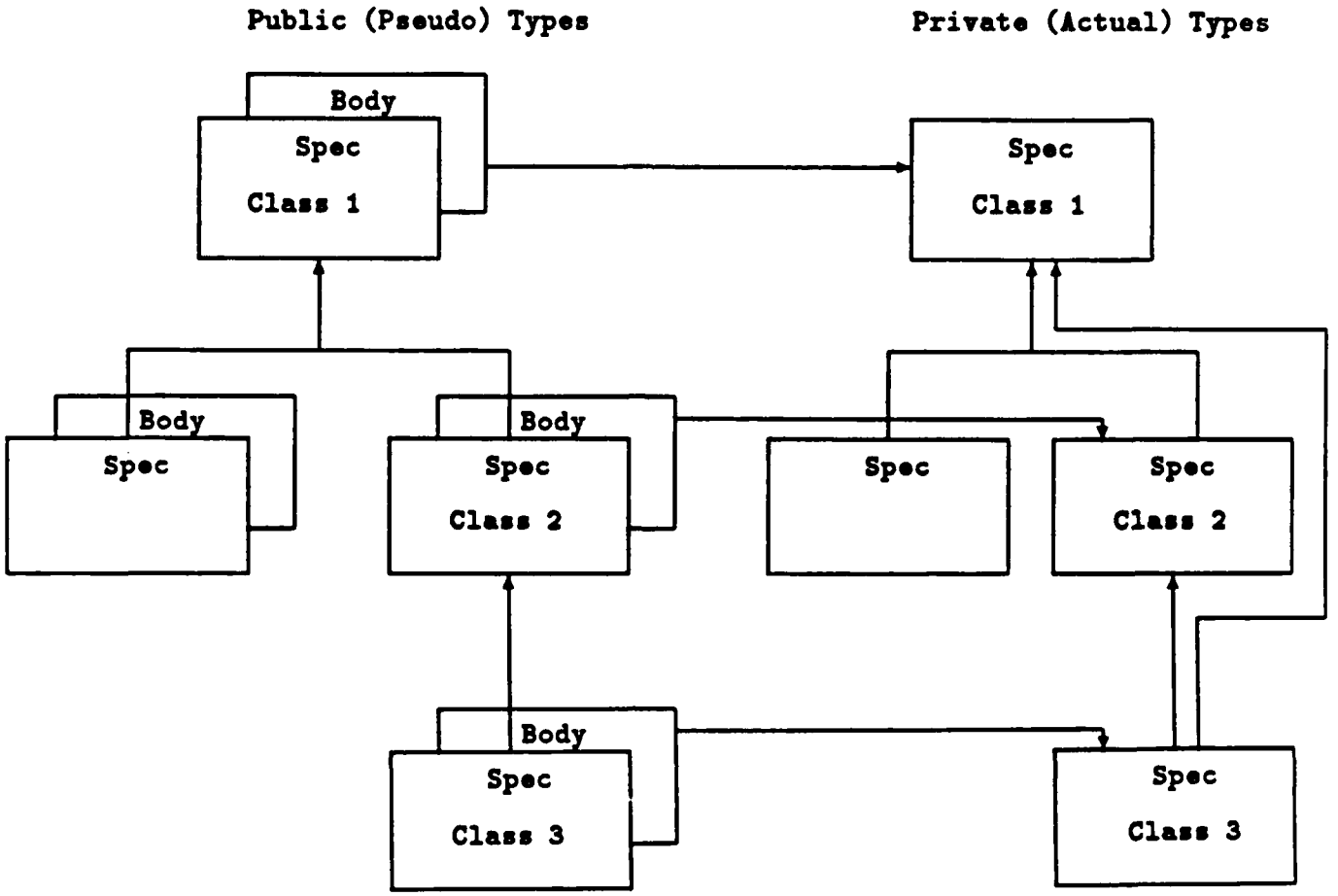
The Ada packaging scheme described in figure 10 has some interesting consequences conerning order of elaboration. In short, the widget type taxonomy must be elaborated in superclass to subclass order; this can be enforced through use of the pre-defined Ada pragma, *elaborate*. Appendix C of this report illustrates the current UR-20 Ada/Xt implementation's widget packaging scheme by providing the full package specification and implementation for the (opaque) widget type, *simple_widget*.

3.4.3 Subprogram and Resource Inheritance

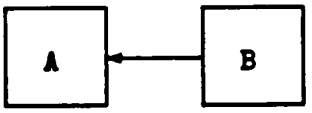
The *actual* type hierarchy provides the data structures to support inheritance, but not the full implementation. Inheritance of subprograms and resources (i.e., widget data fields accessible to the application programmer by named reference) is performed once per widget class, at run-time, in the Ada/Xt implementation. Although this is an implementation detail, it has some impact on the way widget programmers specify widget data structures.

Each time a widget is created (using *xt_create_widget*) the intrinsics check to see that the

[3] An additional "with" dependency exists between the body of the public implementation and the private specification for implementation purposes.



Legend



B ''withs'' A

Figure 10: Widget Types – Ada/Xt Packaging Convention

widget class of the newly created widget has been initialized. If it has not been initialized, the intrinsics invokes a class initialization procedure which passes, in superclass-to-subclass order, the uninitialized widget class *actual* data structure to the class initialization procedure defined for all widget classes (defined as the *class_part_initialize* operation in core_class).

Each class_part_initialize operation is responsible for performing subprogram inheritance (besides other class initialization actions) of subprograms ("methods") defined for the superclass. The class_part_initialize code examines the procedure type fields of the subclass structure passed to it. If any of these fields have special values (called "inherit" values), the class_part_initialize operation will overwrite these fields with the subprogram reference being inherited from the superclass. Thus, it is the responsibility of the widget programmer to request inheritance of superclass methods by use of specially defined constants; it is also the responsibility of the widget programmer to implement the class_part_initialize operation to correctly implement inheritance of inheritable methods defined by widget classes.

Inheritance of resources is managed by the intrinsics (i.e., it is more fully automated). The implementation of resource inheritance is similar to method inheritance: it is done during a one-time initialization of a widget class via a superclass-to-subclass chaining. However, instead of calling widget-specific initialization code, the intrinsics performs a resource list merging and compilation process. The result is that each widget class instance has a list with its resources, and the resources of all of its superclasses. The list is ordered in a subclass-to superclass fashion, so that subclasses may "override" inherited resources. Thus, it is the responsibility of the widget programmer to specify a list of resources which the intrinsics will then "compile" and merge with other lists at class initialization time. This merging is a run-time optimization which bypasses the need for inheritance searches for referenced resources.

4 The Ada/Xt Design

The design approach described in the previous section allows the Ada specification of data types and functional interface to follow the C specification quite closely. The following sections describing the Ada specifications for the Xt Intrinsics follow the C specifications described in [3]. Each section corresponds to a similar chapter in [3], and the specifications should be read in conjunction with [3]. Where the semantics of the subprograms differ from the C version, the differences are noted, otherwise the semantics are as described in [3].

Obviously, there are differences between the Ada specification and the C specification. Most of the differences can be categorized into the following categories:

- C functions with side effects
- pointers versus *out* parameters
- length for list parameters

- argument list assignment
- representation of resource lists
- procedure types

Any C function returning a value and having side effects on parameters has been changed to an Ada *procedure* with one additional argument whose type is the return value type. Since C does not allow *out* parameters to functions, pointers are used. Wherever possible *out* parameters of the base type are used instead of pointers to the base type. Ada provides the *'length* attribute for arrays which can be used instead of supplying a count or length for arrays passed as parameters. The C count parameters are removed wherever possible. The next two differences are closely related to resource management and are discussed at length in the resource management section.

Procedure type specification in Ada is described in the design approach, and all procedure type specifications are provided as *pseudo type* declarations at the end of each section. These *pseudo type* declarations represent instantiations of package *templates*. The following is the template for procedure type packages:

```
package <procedure type name>_procs is

        type <procedure type name>_proc_rep is limited private;
        type <procedure type name>_proc is access xt_callback_proc_rep;

        function xt_inherit_<procedure type name>
                                        return <procedure_type_name>;

        procedure call (the_proc_id : <procedure type name>;
                        <the procedure arguments>);
        generic
           proc_id : in out <procedure type name>_proc;
           with procedure the_proc (<the procedure arguments>);
        package procedure_pointer is
        end procedure_pointer;

private
        -- procedure_control_block is implementation defined
        type <procedure type name>_proc_rep is new procedure_control_block;
end <procedure type name>_procs;
```

Procedure type package definitions appear in the specification as:

```
pseudo_type <procedure type name> is
                new proc_type(<the procedure arguments>);
```

For example, the procedure type *xt_widget_class_proc*, which is a procedure with a single widget_class argument, is defined as:

```
pseudo_type xt_widget_class_proc is
                new proc_type(wc: widget_class);
```

A number of fundamental data types are implementation dependent, and are noted in the specifications. Additional types are defined as private, but left unspecified. These represent opaque data types such as translation tables and resource databases and are not defined in these specifications. The specifications use data types defined in the Ada Xlib bindings. These are contained in the *x_windows* package specification and are not included in this document.

Packaging Considerations

In the following sections, the types and subprogram interfaces which comprise the system-independent Ada/X Toolkit specification are defined as a series of packages. These packages encapsulate groups of related types and operations, and in many cases correspond exactly with groups of related operations as defined in [3]. However, the packaging structure defined in the following sections should be viewed as a guideline for implementing conformant Ada/Xt implementations; alternative packaging models may be desirable or even necessary under some circumstances.

For example, the UR20-UI Ada/Xt *implementation* defines the pre-defined widget and widget classes in separately compiled Ada packages. This packaging model makes the core widget definitions dependent upon the intrinsics package, which defines the type marks for *widget_class* and *widget*. This packaging scheme relies upon the implicit assumption that there exists a mechanism to perform type conversions between objects of type e.g., *widget* – implemented in UR20's implementation as a *system.address* – to objects of type *core_widget*. Should this assumption prove invalid on a given architecture (e.g., an architecture with two addressing modes, such as 32 and 48 bit addressing modes, might implement system.address as a variant record), alternative implementations may impose alternative packaging models. For example, in the above multi-address type architecture, the pre-defined *core* type could be defined in a subpackage of the intrinsics package; then type *widget* could be defined as an access to type *core*. This would bypass the need to convert system address types to widget types [4].

The following specifications try to strike a balance between sufficient conciseness and overspecification. Specific type representations or packaging decisions should be considered as very strong recommendations. For example, the decision to represent a list type as an unconstrained array of some base type should be considered part of the system-independent specification; also, the decision to include a type within a package is a similar "strong recommendation." In many cases, however, type definitions are not given a package context; in these cases, implementations should consider themselves free to determine their own packaging.

Figure 11 provides a top-level overview of the packaging structure for the UR20-UI

[4] Although it would still be necessary to perform type conversions between *distinct* access types.

Ada/Xt *implementation.* The figure does not include all of the pre-defined widget type dependencies; those that are included are representative of the packaging model.

Note that the following specifications assume a foundational set of X types, as for example defined in the SAIC STARS Foundations Ada/Xlib bindings. In the following specifications, such types are prefixed by "x_windows.*type*," even though the Ada/Xlib bindings may require deeper qualification to subpackages. Types that are not preceded by the x_windows package name can be assumed to be either intrinsic types to Ada, or defined by Ada/Xt.

Finally, note that some types are referenced before they are defined. This is necessary in order to associate type declarations with appropriate operations.

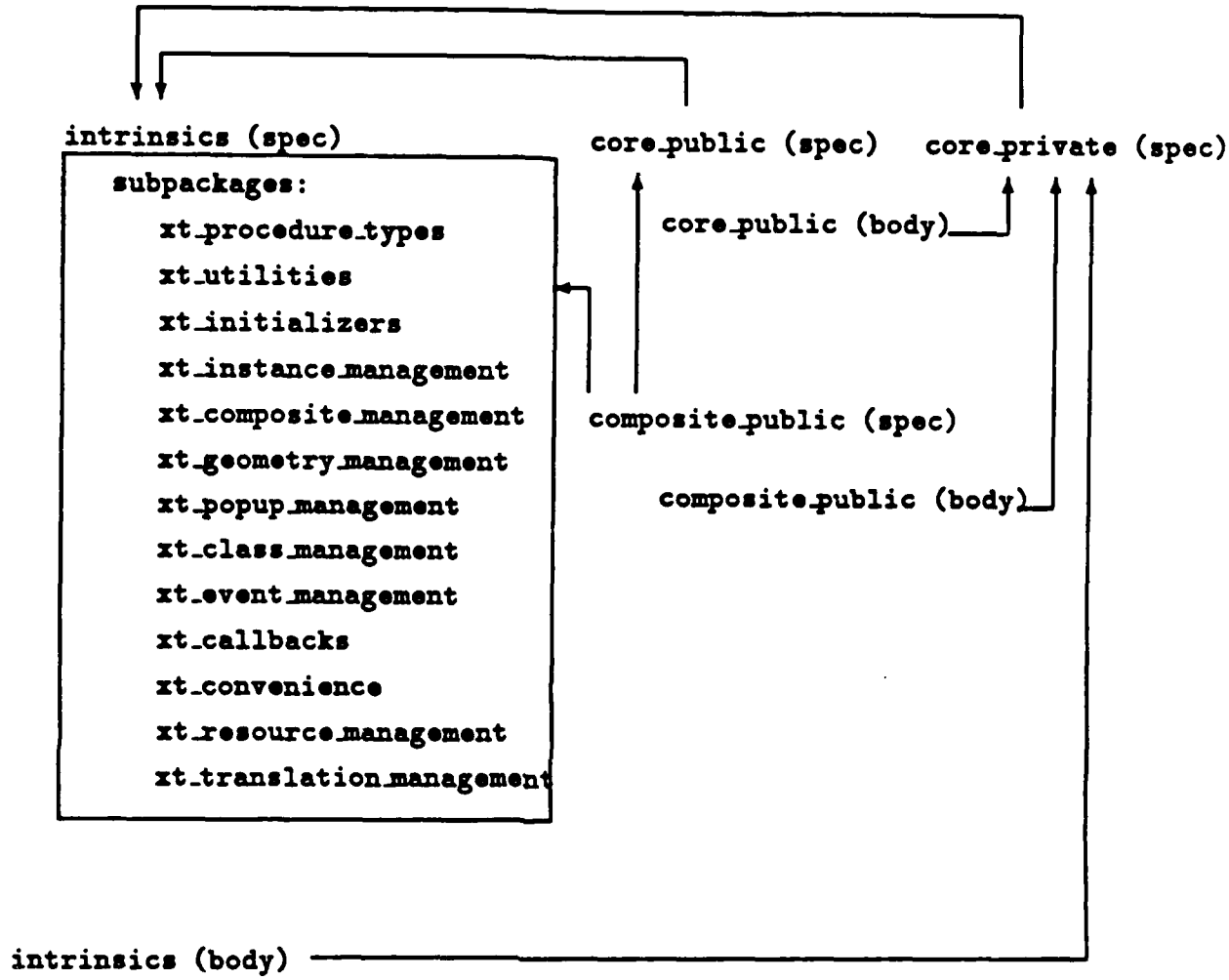


Figure 11: UR20-UI Ada/Xt Packaging Implementation

4.1 Widgets

The basic abstraction in the toolkit is the widget and its associated widget class. The three basic widget types

1. Core
2. Composite
3. Constraint

are described here. The structures representing these three widget types map directly to the C data structures. See 1.3 of [3] for a complete discussion of widgets. All widgets are derived types of an implementation defined type *widget* or subclasses of *widget*, and widget classes are subtypes of an implementation defined type *widget_class* or subclasses of type *widget_class*. *widget* and *widget_class* are usually some form of physical address to the data structures described in this section. The intrinsics provide conversion routines for all the widget and widget class types known to the intrinsics. The default values for the widget types described here are the same as specified in 1.3 of [3].

4.1.1 Core Widgets

The following types are assumed to be visible to the core_private package specification, and are not defined elsewhere in this report:

```
type widget is implementation_defined;
type widget_class is implementation_defined;
type widget_list is array (natural range <>) of widget;
type widget_list_ptr is access widget_list;

type cardinal is range 0 .. implementation_defined;
subtype position is cardinal;
subtype dimension is cardinal;

subtype xt_offset is cardinal;
type xt_offset_list is array (natural range <>) of xt_offset;
type xt_offset_list_ptr is access xt_offset_list;

type xt_string is access string;

type xt_boolean is implementation_defined;
type xt_version_type is implementation_defined;
```

The following package defines the pre-defined core widget class and instance types:

```
package core_private is

--1) define widget_part (core is special case -- no nested records)

   core_part_size : constant cardinal := implementation_defined;
   type core_part is record
      self : widget;
      widgetclass : widget_class;
      parent : widget;
      the_xrm_name: x_windows.xrm_name;
      being_destroyed : xt_boolean;
      destroy_callbacks: xt_callback_list_ptr;
      constraints : x_windows.caddr_t;
      x : position;
      y : position;
      width : dimension;
      height: dimension;
      border_width : dimension;
      managed : xt_boolean;
      sensitive: xt_boolean;
      ancestor_sensitive : xt_boolean;
      event_table : xt_event_table;
      tm : xt_TM_Rec;
      accelerators : xt_translations;
      border_pixel : x_windows.pixel;
      border_pixmap : x_windows.pixmap;
      popup_list : widget_list_ptr;
      name : xt_string;
      my_screen : x_windows.screen;
      my_colormap: x_windows.color_map;
      my_window : x_windows.window;
      depth : cardinal;
      background_pixel : x_windows.pixel;
      background_pixmap : x_windows.pixmap;
      visible : xt_boolean;
      mapped_when_managed : xt_boolean;
   end record;

   type core_part_pointer is access core_part;
   type core_widget_pointer is access core_part;

-- 2) define class part

   core_class_part_size : constant cardinal := implementation_defined;
   type core_class_part is record
      superclass : widget_class;
      class_name : xt_string;
      widget_size: cardinal;
      class_initialize : xt_proc;
      class_part_initialize : xt_widget_class_proc;
      class_inited : xt_boolean;
      initialize : xt_init_proc;
```

```
      initialize_hook : xt_args_proc;
      realize : xt_realize_proc;
      actions : xt_action_list_ptr;
      resources: xt_resource_list_ptr;
      the_xrm_class: x_windows.xrm_class;
      compress_motion : xt_boolean;
      compress_exposure : xt_boolean;
      compress_interleave : xt_boolean;
      visible_interest : xt_boolean;
      destroy : xt_widget_proc;
      resize  : xt_widget_proc;
      expose  : xt_expose_proc;
      set_values : xt_set_values_func;
      set_values_hook : xt_args_func;
      set_values_almost : xt_almost_proc;
      get_values_hook : xt_args_proc;
      accept_focus : xt_accept_focus_proc;
      version : xt_version_type;
      callback_private : xt_offset_list_ptr;
      tm_table : xt_string;
      query_geometry : xt_geometry_handler;
      display_accelerator : xt_string_proc;
      extension : x_windows.caddr_t;
   end record;

-- types for conversion operations:

   type core_class_part_pointer is access core_class_part;
   type core_class_pointer is access core_class_part;

-- allocate the class constant

   function to_widget_class is new unchecked_conversion(
      source => core_class_pointer,
      target => widget_class);

   the_core_class : constant widget_class :=
      to_widget_class (new core_class_part);

end core_private;
```

4.1.2 Composite Widgets

```
-- superclass context
with core_private; use core_private;
package composite_private is

   composite_part_rec_size : constant cardinal := implementation_defined;
   type composite_part_rec is record
      children : widget_list_ptr;
      num_slots : cardinal;
```

```
      insert_position : xt_order_proc;
   end record;

   composite_widget_size : constant cardinal :=
      core_part_size + composite_part_rec_size;
   type composite_widget_rec is record
      core_part : core_private.core_part;
      composite_part : composite_part_rec;
   end record;

   for composite_widget_rec use record at mod implementation_defined;
      core_part
         at 0
         range 0 .. core_part_size - 1;
      composite_part
         at 0
         range core_part_size .. core_part_size + composite_part_rec_size - 1;
   end record;

   type composite_part_pointer is access composite_part_rec;
   type composite_widget_pointer is access composite_widget_rec;

   composite_class_part_rec_size : constant cardinal := implementation_defined;
   type composite_class_part_rec is record
      geometry_handler : xt_geometry_handler;
      change_managed : xt_widget_proc;
      insert_child : xt_widget_proc;
      delete_child : xt_widget_proc;
      extension : x_windows.caddr_t;
   end record;

   composite_class_part_size : constant cardinal :=
      core_class_part_size + composite_class_part_rec_size;
   type composite_class_part is record
      core_class_part : core_private.core_class_part;
      composite_class_part : composite_class_part_rec;
   end record;

   for composite_class_part use record at mod implementation_defined;
      core_class_part
         at 0
         range 0 .. core_class_part_size - 1;
      composite_class_part
         at 0
         range core_class_part_size .. composite_class_part_size - 1;
   end record;

   type composite_class_part_pointer is access composite_class_part_rec;
   type composite_class_pointer is access composite_class_part;

-- allocate the class constant
```

```
   function to_widget_class is new unchecked_conversion(
      source => composite_class_pointer,
      target => widget_class);

   the_composite_class : constant widget_class :=
      to_widget_class (new composite_class_part);

end composite_private;
```

4.1.3 Constraint Widgets

```
-- superclass context
with composite_private; use composite_private;
with core_private; use core_private;

package constraint_private is

   constraint_part_rec_size : constant cardinal := 0;
   type constraint_part_rec is record
      null;
   end record;

   constraint_widget_size : constant cardinal :=
      composite_widget_size + constraint_part_rec_size;
   type constraint_widget_rec is record
      core_part : core_private.core_part;
      composite_part : composite_private.composite_part_rec;
      constraint_part : constraint_part_rec;
   end record;

   for constraint_widget_rec use record at mod implementation_defined;
      core_part
         at 0
         range 0 .. core_part_size - 1;
      composite_part
         at 0
         range core_part_size .. composite_widget_size - 1;
      constraint_part
         at 0
         range composite_widget_size .. constraint_widget_size - 1;
   end record;

   type constraint_part_pointer is access constraint_part_rec;
   type constraint_widget_pointer is access constraint_widget_rec;

   constraint_class_part_rec_size : constant cardinal := implementation_defined;
   type constraint_class_part_rec is record
      resources : xt_resource_list_ptr;
      constraint_size : cardinal;
      initialize : xt_init_proc;
      destroy : xt_widget_proc;
      set_values : xt_set_values_func;
```

```
      extension : x_windows.caddr_t;
   end record;

   constraint_class_part_size : constant cardinal :=
      composite_class_part_size + constraint_class_part_rec_size;
   type constraint_class_part is record
      core_class_part : core_private.core_class_part;
      composite_class_part : composite_private.composite_class_part_rec;
      constraint_class_part : constraint_class_part_rec;
   end record;

   for constraint_class_part use record at mod implementation_defined;
      core_class_part
         at 0
         range 0 .. core_class_part_size - 1;
      composite_class_part
         at 0
         range core_class_part_size .. composite_class_part_size - 1;
      constraint_class_part
         at 0
         range composite_class_part_size .. constraint_class_part_size - 1;
   end record;

   type constraint_class_part_pointer is access constraint_class_part_rec;
   type constraint_class_pointer is access constraint_class_part;

-- allocate the class constant

   function to_widget_class is new unchecked_conversion(
      source => constraint_class_pointer,
      target => widget_class);

   the_constraint_class : constant widget_class :=
      to_widget_class (new constraint_class_part);

end constraint_private;
```

4.1.4 Widget Class and Superclass Look Up

```
function xt_class (w : widget) return widget_class;
function xt_superclass (w : widget) return widget_class;
function xt_is_subclass (w  : widget;
                         wc : widget_class) return boolean;
procedure xt_check_subclass (w       : widget;
                             wc      : widget_class;
                             message : string);
```

4.2 Widget Instantiation

This section describes widget instantiation. Refer to chapter 2 of [3] for a complete description of widget instantiation. The following is the Ada programmatic interface to functions and data structures required for widget instantiation.

4.2.1 Toolkit Initialization

The following types are assumed to be visible to xt_initializers, and may be declared within this package:

```
type application_context is private;
```

The package specification *xt_initializers* specifies subprograms and data structures used in toolkit initialization.

```
package xt_initializers is

   procedure xt_toolkit_initialize;

   function xt_create_application_context return application_context;

   procedure xt_destroy_application_context
          (context : in out application_context);

   function xt_widget_to_application_context
          (w : widget) return application_context;

   procedure xt_display_initialize (app_context       : application_context;
                                    the_display       : x_windows.display;
                                    application_name  : string;
                                    application_class : string;
                                    options : xrm_option_desc_rec_list;
                                    argc              : in out cardinal;
                                    argv              : in out string);

   procedure xt_open_display (app_context       : application_context;
                              display_string    : string;
                              application_name  : string;
                              application_class : string;
                              options           : xrm_option_desc_rec_list;
                              argc              : in out cardinal;
                              argv              : in out string;
                              return_display    : out x_windows.display);

   procedure xt_close_display (the_display : in out x_windows.display);

end xt_initializers;
```

4.2.2 Loading the Resource Database

```
function xt_database (the_display : x_windows.display) return
   x_windows.xrm_database;
```

4.2.3 Parsing the Command Line

Although Ada compilers differ in handling the Unix notion of *argc* and *argv*, the Ada Toolkit recognizes the standard X command line options. The type definitions for command line option description records are:

```
type xrm_option_kind is (xrm_option_no_arg,
                         xrm_option_is_arg,
                         xrm_option_sticky_arg,
                         xrm_option_set_arg,
                         xrm_option_res_arg,
                         xrm_option_skip_arg,
                         xrm_option_skip_line);

type xrm_option_desc_rec is record
    option         : xt_string;
    resource_name  : xt_string;
    arg_kind       : xrm_option_kind;
    value          : x_windows.caddr_t;
end record;
type xrm_option_desc_list is
          array (Natural range <>) of xrm_option_desc_rec;
```

4.2.4 Creating Widgets

Widget creation in Ada differs only in its treatment of argument lists. Argument lists in C are essentially lists of untyped data; a problem for strongly typed languages like Ada. Since argument lists are closely related to resource management, a discussion of the handling of these lists in Ada is deferred to the section on resource management.

The following types are assumed to be visible to package xt_instance_management, and can be defined within this package:

```
subtype xt_arg_val isx_windows.caddr_t;
type xt_arg is  record
         name  : xt_string;
         value : xt_arg_val;
end record;
type arg_list is array (natural range <>) of xt_arg;
type arg_list_ptr is access arg_list;
```

The following package declaration provides basic instance manipulation primitives:

```
package xt_instance_management is
```

```
   function xt_create_widget (name     : string;
                              of_class : widget_class;
                              parent   : widget;
                              args     : arg_list) return widget;

   function xt_app_create_shell (application_name  : string;
                                 application_class : string;
                                 wc                : widget_class;
                                 the_display       : x_windows.display;
                                 args              : arg_list)
                                    return widget;

   procedure xt_add_callback (w             : widget;
                              callback_name : string;
                              callback      : xt_callback_proc;
                              client_data   : x_windows.caddr_t);

   procedure xt_create_window (w          : widget;
                               win_class  : x_windows.window_class;
                               the_visual : x_windows.visual;
                               value_mask : xt_value_mask;
                               attributes : x_windows.x_set_window_attributes);

   procedure xt_realize_widget (w : widget);

   procedure xt_unrealize_widget (w : widget);

   procedure xt_destroy_widget (w : widget);

   function xt_is_realized (w : widget) return boolean;

   function xt_display (w : widget) return x_windows.display;

   function xt_parent (w : widget) return widget;

   function xt_screen (w : widget) return x_windows.screen;

   function xt_window (w : widget) return x_windows.window;

end xt_instance_management;
```

4.3 Composite Widget Management

The package specification *xt_composite_management* specifies the subprogram units providing functions for managing children of composite widgets. These subprograms are described in chapter 3 of [3].

```
package xt_composite_management is

   function xt_is_composite (w : widget) return boolean;

   procedure xt_manage_children (wlist : widget_list);
```

```
   procedure xt_manage_child (child : widget);

   procedure xt_unmanage_children (wlist : widget_list);

   procedure xt_unmanage_child (wlist : widget_list);

   function xt_is_managed (w : widget) return boolean;

   function xt_create_managed_widget (name     : string;
                                      of_class : widget_class;
                                      parent   : widget;
                                      args     : arg_list) return widget;

   procedure xt_set_mapped_when_managed (w               : in widget;
                                         map_when_manged : boolean := true);

end xt_composite_management;
```

4.3.1 Procedure Types in Composite Widgets

The following package specification describes the procedure type for the *insert_child* and *delete_child* procedures used for adding/deleting children of a composite widget.

```
pseudo_type xt_widget_proc is
                  new proc_type(the_widget : in out widget);
```

The procedure type for specifying the insertion order of children, the *insert_position* field of a composite widget, is:

```
pseudo_type xt_order_proc is
                  new proc_type(the_widget  : in out widget);
```

4.4 Shell Widgets

The following data structures specify the various shell widgets described in chapter 4 of [3]. The shell widgets are:

- Shell
- Override Shell
- WM Shell
- Vendor Shell
- Transient Shell
- Top Level Shell

- Application Shell

```
-- superclass context:
with composite_private; use composite_private;
with core_private; use core_private;

package shell_private is

   -- Define various shell parts and widget record extensions

   -- shell_widget

   shell_part_rec_size : constant cardinal := implementation_defined;
   type shell_part_rec is record
      geometry : xt_string;
      create_child_popup_proc : xt_proc;
      grab_kind : xt_grab_kind;
      spring_loaded : xt_boolean;
      popped_up : xt_boolean;
      allow_shell_resize : xt_boolean;
      client_specified : xt_boolean;
      save_under : xt_boolean;
      override_redirect : xt_boolean;
      popup_callback : xt_callback_list_ptr;
      popdown_callback : xt_callback_list_ptr;
   end record;

   shell_widget_size : constant cardinal :=
      composite_widget_size + shell_part_rec_size;
   type shell_widget_rec is record
      core_part : core_private.core_part;
      composite_part : composite_private.composite_part_rec;
      shell_part : shell_part_rec;
   end record;

   for shell_widget_rec use record at mod implementation_defined;
      core_part
      at 0
      range 0 .. core_part_size - 1;
      composite_part
      at 0
      range core_part_size .. composite_widget_size - 1;
      shell_part
         at 0
         range composite_widget_size .. shell_widget_size - 1;
   end record;

   type shell_part_pointer is access shell_part_rec;
   type shell_widget_pointer is access shell_widget_rec;

   -- override_shell_widget

   override_shell_part_rec_size : constant cardinal := implementation_defined;
```

```
type override_shell_part_rec is record
   null;
end record;

override_shell_widget_size : constant cardinal :=
   shell_widget_size + override_shell_part_rec_size;
type override_shell_widget_rec is record
   core_part : core_private.core_part;
   composite_part : composite_private.composite_part_rec;
   shell_part : shell_part_rec;
   override_shell_part : override_shell_part_rec;
end record;

for override_shell_widget_rec use record at mod implementation_defined;
   core_part
   at 0
   range 0 .. core_part_size - 1;
   composite_part
   at 0
   range core_part_size .. composite_widget_size - 1;
   shell_part
      at 0
      range composite_widget_size .. shell_widget_size - 1;
   override_shell_part
      at 0
      range shell_widget_size .. override_shell_widget_size - 1;
end record;

type override_shell_part_pointer is access override_shell_part_rec;
type override_shell_widget_pointer is access override_shell_widget_rec;

-- wm_shell_widget

wm_shell_part_rec_size : constant cardinal := implementation_defined;
type wm_shell_part_rec is record
   title : xt_string;
   wm_timeout : x_windows.time;
   wait_for_wm : xt_boolean;
   transient : xt_boolean;
   size_hints : x_windows.caddr_t;
   wm_hints : x_windows.caddr_t;
end record;

wm_shell_widget_size : constant cardinal :=
   override_shell_widget_size + wm_shell_part_rec_size;
type wm_shell_widget_rec is record
   core_part : core_private.core_part;
   composite_part : composite_private.composite_part_rec;
   shell_part : shell_part_rec;
   override_shell_part : override_shell_part_rec;
   wm_shell_part : wm_shell_part_rec;
end record;
```

```
for wm_shell_widget_rec use record at mod implementation_defined;
   core_part
   at 0
   range 0 .. core_part_size - 1;
   composite_part
   at 0
   range core_part_size .. composite_widget_size - 1;
   shell_part
      at 0
      range composite_widget_size .. shell_widget_size - 1;
   override_shell_part
      at 0
      range shell_widget_size .. override_shell_widget_size - 1;
   wm_shell_part
      at 0
      range override_shell_widget_size .. wm_shell_widget_size - 1;
end record;

type wm_shell_part_pointer is access wm_shell_part_rec;
type wm_shell_widget_pointer is access wm_shell_widget_rec;

-- vendor_shell_widget

vendor_shell_part_rec_size : constant cardinal := implementation_defined;
type vendor_shell_part_rec is record
   vendor_specific : implementation_defined;
end record;

vendor_shell_widget_size : constant cardinal :=
   wm_shell_widget_size + vendor_shell_part_rec_size;
type vendor_shell_widget_rec is record
   core_part : core_private.core_part;
   composite_part : composite_private.composite_part_rec;
   shell_part : shell_part_rec;
   override_shell_part : override_shell_part_rec;
   wm_shell_part : wm_shell_part_rec;
   vendor_shell_part : vendor_shell_part_rec;
end record;

for vendor_shell_widget_rec use record at mod implemetation_defined;
   core_part
   at 0
   range 0 .. core_part_size - 1;
   composite_part
   at 0
   range core_part_size .. composite_widget_size - 1;
   shell_part
      at 0
      range composite_widget_size .. shell_widget_size - 1;
   override_shell_part
      at 0
```

```
      range shell_widget_size .. override_shell_widget_size - 1;
   wm_shell_part
      at 0
      range override_shell_widget_size .. wm_shell_widget_size - 1;
   vendor_shell_part
      at 0
      range wm_shell_widget_size .. vendor_shell_widget_size - 1;
end record;

type vendor_shell_part_pointer is access vendor_shell_part_rec;
type vendor_shell_widget_pointer is access vendor_shell_widget_rec;

-- transient_shell_widget

transient_shell_part_rec_size : constant cardinal := 0;
type transient_shell_part_rec is record
   null;
end record;

transient_shell_widget_size : constant cardinal :=
   vendor_shell_widget_size + transient_shell_part_rec_size;
type transient_shell_widget_rec is record
   core_part : core_private.core_part;
   composite_part : composite_private.composite_part_rec;
   shell_part : shell_part_rec;
   override_shell_part : override_shell_part_rec;
   wm_shell_part : wm_shell_part_rec;
   vendor_shell_part : vendor_shell_part_rec;
   transient_shell_part : transient_shell_part_rec;
end record;

for transient_shell_widget_rec use record at mod implementation_defined;
   core_part
   at 0
   range 0 .. core_part_size - 1;
   composite_part
   at 0
   range core_part_size .. composite_widget_size - 1;
   shell_part
      at 0
      range composite_widget_size .. shell_widget_size - 1;
   override_shell_part
      at 0
      range shell_widget_size .. override_shell_widget_size - 1;
   wm_shell_part
      at 0
      range override_shell_widget_size .. wm_shell_widget_size - 1;
   vendor_shell_part
      at 0
      range wm_shell_widget_size .. vendor_shell_widget_size - 1;
   transient_shell_part
      at 0
```

```
      range vendor_shell_widget_size .. transient_shell_widget_size - 1;
end record;

type transient_shell_part_pointer is access transient_shell_part_rec;
type transient_shell_widget_pointer is access transient_shell_widget_rec;

-- top_level_shell_widget

top_level_shell_part_rec_size : constant cardinal := implementation_defined;
type top_level_shell_part_rec is record
   icon_name : xt_string;
   iconic : xt_boolean;
end record;

top_level_shell_widget_size : constant cardinal :=
   transient_shell_widget_size + top_level_shell_part_rec_size;
type top_level_shell_widget_rec is record
   core_part : core_private.core_part;
   composite_part : composite_private.composite_part_rec;
   shell_part : shell_part_rec;
   override_shell_part : override_shell_part_rec;
   wm_shell_part : wm_shell_part_rec;
   vendor_shell_part : vendor_shell_part_rec;
   transient_shell_part : transient_shell_part_rec;
   top_level_shell_part : top_level_shell_part_rec;
end record;

for top_level_shell_widget_rec use record at mod implementation_defined;
   core_part
   at 0
   range 0 .. core_part_size - 1;
   composite_part
   at 0
   range core_part_size .. composite_widget_size - 1;
   shell_part
      at 0
      range composite_widget_size .. shell_widget_size - 1;
   override_shell_part
      at 0
      range shell_widget_size .. override_shell_widget_size - 1;
   wm_shell_part
      at 0
      range override_shell_widget_size .. wm_shell_widget_size - 1;
   vendor_shell_part
      at 0
      range wm_shell_widget_size .. vendor_shell_widget_size - 1;
   transient_shell_part
      at 0
      range vendor_shell_widget_size .. transient_shell_widget_size - 1;
   top_level_shell_part
      at 0
      range transient_shell_widget_size .. top_level_shell_widget_size - 1;
```

```
end record;

type top_level_shell_part_pointer is access top_level_shell_part_rec;
type top_level_shell_widget_pointer is access top_level_shell_widget_rec;

-- application_shell_widget

application_shell_part_rec_size :
   constant cardinal := implementation_defined;
type application_shell_part_rec is record
   class : widget_class;
   the_xrm_class : x_windows.xrm_class;
   argc : cardinal;
   argv : argv_type;
end record;

application_shell_widget_size : constant cardinal :=
   top_level_shell_widget_size + application_shell_part_rec_size;
type application_shell_widget_rec is record
   core_part : core_private.core_part;
   composite_part : composite_private.composite_part_rec;
   shell_part : shell_part_rec;
   override_shell_part : override_shell_part_rec;
   wm_shell_part : wm_shell_part_rec;
   vendor_shell_part : vendor_shell_part_rec;
   transient_shell_part : transient_shell_part_rec;
   top_level_shell_part : top_level_shell_part_rec;
   application_shell_part : application_shell_part_rec;
end record;

for application_shell_widget_rec use record at mod implementation_defined;
   core_part
   at 0
   range 0 .. core_part_size - 1;
   composite_part
   at 0
   range core_part_size .. composite_widget_size - 1;
   shell_part
      at 0
      range composite_widget_size .. shell_widget_size - 1;
   override_shell_part
      at 0
      range shell_widget_size .. override_shell_widget_size - 1;
   wm_shell_part
      at 0
      range override_shell_widget_size .. wm_shell_widget_size - 1;
   vendor_shell_part
      at 0
      range wm_shell_widget_size .. vendor_shell_widget_size - 1;
   transient_shell_part
      at 0
      range vendor_shell_widget_size .. transient_shell_widget_size - 1;
```

```
      top_level_shell_part
         at 0
         range transient_shell_widget_size .. top_level_shell_widget_size - 1;
      application_shell_part
         at 0
         range top_level_shell_widget_size ..
               application_shell_widget_size - 1;
   end record;

   type application_shell_part_pointer is access application_shell_part_rec;
   type application_shell_widget_pointer is access
                                              application_shell_widget_rec;

   -- define class records for shell widget classes

   -- shell_class

   shell_class_part_rec_size : constant cardinal := implementation_defined;
   type shell_class_part_rec is record
      extension : x_windows.caddr_t;
   end record;

   shell_class_part_size : constant cardinal :=
      composite_class_part_size + shell_class_part_rec_size;
   type shell_class_part is record
      core_class_part : core_private.core_class_part;
      composite_class_part : composite_private.composite_class_part_rec;
      shell_class_part : shell_class_part_rec;
   end record;

   for shell_class_part use record at mod implementation_defined;
      core_class_part
      at 0
      range 0 .. core_class_part_size - 1;
      composite_class_part
         at 0
         range core_class_part_size .. composite_class_part_size - 1;
      shell_class_part
      at 0
      range composite_class_part_size .. shell_class_part_size - 1;
   end record;

   type shell_class_part_pointer is access shell_class_part_rec;
   type shell_class_pointer is access shell_class_part;

   -- override_shell_class

   override_shell_class_part_rec_size :
      constant cardinal := implementation_defined;
   type override_shell_class_part_rec is record
      extension : x_windows.caddr_t;
   end record;
```

```
override_shell_class_part_size : constant cardinal :=
   shell_class_part_size + override_shell_class_part_rec_size;
type override_shell_class_part is record
   core_class_part : core_private.core_class_part;
   composite_class_part : composite_private.composite_class_part_rec;
   shell_class_part : shell_class_part_rec;
   override_shell_class_part : override_shell_class_part_rec;
end record;

for override_shell_class_part use record at mod implementation_defined;
   core_class_part
   at 0
   range 0 .. core_class_part_size - 1;
   composite_class_part
      at 0
      range core_class_part_size .. composite_class_part_size - 1;
   shell_class_part
   at 0
   range composite_class_part_size .. shell_class_part_size - 1;
   override_shell_class_part
      at 0
      range shell_class_part_size .. override_shell_class_part_size - 1;
end record;

type override_shell_class_part_pointer is
                                    access override_shell_class_part_rec;
type override_shell_class_pointer is access override_shell_class_part;

-- wm_shell_class

wm_shell_class_part_rec_size : constant cardinal := implementation_defined;
type wm_shell_class_part_rec is record
   extension : x_windows.caddr_t;
end record;

wm_shell_class_part_size : constant cardinal :=
   override_shell_class_part_size + wm_shell_class_part_rec_size;
type wm_shell_class_part is record
   core_class_part : core_private.core_class_part;
   composite_class_part : composite_private.composite_class_part_rec;
   shell_class_part : shell_class_part_rec;
   override_shell_class_part : override_shell_class_part_rec;
   wm_shell_class_part : wm_shell_class_part_rec;
end record;

for wm_shell_class_part use record at mod implementation_defined;
   core_class_part
   at 0
   range 0 .. core_class_part_size - 1;
   composite_class_part
      at 0
```

```
      range core_class_part_size .. composite_class_part_size - 1;
   shell_class_part
   at 0
   range composite_class_part_size .. shell_class_part_size - 1;
   override_shell_class_part
      at 0
      range shell_class_part_size .. override_shell_class_part_size - 1;
   wm_shell_class_part
      at 0
      range override_shell_class_part_size .. wm_shell_class_part_size - 1;
end record;

type wm_shell_class_part_pointer is access wm_shell_class_part_rec;
type wm_shell_class_pointer is access wm_shell_class_part;

-- vendor_shell_class

vendor_shell_class_part_rec_size :
   constant cardinal := implementation_defined;
type vendor_shell_class_part_rec is record
   extension : x_windows.caddr_t;
end record;

vendor_shell_class_part_size : constant cardinal :=
   wm_shell_class_part_size + vendor_shell_class_part_rec_size;
type vendor_shell_class_part is record
   core_class_part : core_private.core_class_part;
   composite_class_part : composite_private.composite_class_part_rec;
   shell_class_part : shell_class_part_rec;
   override_shell_class_part : override_shell_class_part_rec;
   wm_shell_class_part : wm_shell_class_part_rec;
   vendor_shell_class_part : vendor_shell_class_part_rec;
end record;

for vendor_shell_class_part use record at mod implementation_defined;
   core_class_part
   at 0
   range 0 .. core_class_part_size - 1;
   composite_class_part
      at 0
      range core_class_part_size .. composite_class_part_size - 1;
   shell_class_part
   at 0
   range composite_class_part_size .. shell_class_part_size - 1;
   override_shell_class_part
      at 0
      range shell_class_part_size .. override_shell_class_part_size - 1;
   wm_shell_class_part
      at 0
      range override_shell_class_part_size .. wm_shell_class_part_size - 1;
   vendor_shell_class_part
      at 0
```

```
      range wm_shell_class_part_size ..
            vendor_shell_class_part_size - 1;
end record;

type vendor_shell_class_part_pointer is access vendor_shell_class_part_rec;
type vendor_shell_class_pointer is access vendor_shell_class_part;

-- transient_shell_class

transient_shell_class_part_rec_size :
   constant cardinal := implementation_defined;

type transient_shell_class_part_rec is record
   extension : x_windows.caddr_t;
end record;

transient_shell_class_part_size : constant cardinal :=
   vendor_shell_class_part_size + transient_shell_class_part_rec_size;
type transient_shell_class_part is record
   core_class_part : core_private.core_class_part;
   composite_class_part : composite_private.composite_class_part_rec;
   shell_class_part : shell_class_part_rec;
   override_shell_class_part : override_shell_class_part_rec;
   wm_shell_class_part : wm_shell_class_part_rec;
   vendor_shell_class_part : vendor_shell_class_part_rec;
   transient_shell_class_part : transient_shell_class_part_rec;
end record;

for transient_shell_class_part use record at mod implementation_defined;
   core_class_part
   at 0
   range 0 .. core_class_part_size - 1;
   composite_class_part
      at 0
      range core_class_part_size .. composite_class_part_size - 1;
   shell_class_part
   at 0
   range composite_class_part_size .. shell_class_part_size - 1;
   override_shell_class_part
      at 0
      range shell_class_part_size .. override_shell_class_part_size - 1;
   wm_shell_class_part
      at 0
      range override_shell_class_part_size .. wm_shell_class_part_size - 1;
   vendor_shell_class_part
      at 0
      range wm_shell_class_part_size ..
            vendor_shell_class_part_size - 1;
   transient_shell_class_part
      at 0
      range vendor_shell_class_part_size ..
            transient_shell_class_part_size - 1;
```

```
end record;

type transient_shell_class_part_pointer is
                                    access transient_shell_class_part_rec;
type transient_shell_class_pointer is
                                       access transient_shell_class_part;

-- top_level_shell_class

top_level_shell_class_part_rec_size :
   constant cardinal := implementation_defined;
type top_level_shell_class_part_rec is record
   extension : x_windows.caddr_t;
end record;

top_level_shell_class_part_size : constant cardinal :=
   transient_shell_class_part_size + top_level_shell_class_part_rec_size;
type top_level_shell_class_part is record
   core_class_part : core_private.core_class_part;
   composite_class_part : composite_private.composite_class_part_rec;
   shell_class_part : shell_class_part_rec;
   override_shell_class_part : override_shell_class_part_rec;
   wm_shell_class_part : wm_shell_class_part_rec;
   vendor_shell_class_part : vendor_shell_class_part_rec;
   transient_shell_class_part : transient_shell_class_part_rec;
   top_level_shell_class_part : top_level_shell_class_part_rec;
end record;

for top_level_shell_class_part use record at mod implementation_defined;
   core_class_part
   at 0
   range 0 .. core_class_part_size - 1;
   composite_class_part
      at 0
      range core_class_part_size .. composite_class_part_size - 1;
   shell_class_part
   at 0
   range composite_class_part_size .. shell_class_part_size - 1;
   override_shell_class_part
      at 0
      range shell_class_part_size .. override_shell_class_part_size - 1;
   wm_shell_class_part
      at 0
      range override_shell_class_part_size .. wm_shell_class_part_size - 1;
   vendor_shell_class_part
      at 0
      range wm_shell_class_part_size ..
            vendor_shell_class_part_size - 1;
   transient_shell_class_part
      at 0
      range vendor_shell_class_part_size ..
            transient_shell_class_part_size - 1;
```

```
      top_level_shell_class_part
         at 0
         range transient_shell_class_part_size ..
               top_level_shell_class_part_size - 1;
   end record;

   type top_level_shell_class_part_pointer is
                                    access top_level_shell_class_part_rec;
   type top_level_shell_class_pointer is
                                       access top_level_shell_class_part;


   -- application_shell_class

   application_shell_class_part_rec_size :
      constant cardinal := implementation_defined;
   type application_shell_class_part_rec is record
      extension : x_windows.caddr_t;
   end record;

      top_level_shell_class_part_size + application_shell_class_part_rec_size;
   type application_shell_class_part is record
      core_class_part : core_private.core_class_part;
      composite_class_part : composite_private.composite_class_part_rec;
      shell_class_part : shell_class_part_rec;
      override_shell_class_part : override_shell_class_part_rec;
      vm_shell_class_part : vm_shell_class_part_rec;
      vendor_shell_class_part : vendor_shell_class_part_rec;
      transient_shell_class_part : transient_shell_class_part_rec;
      top_level_shell_class_part : top_level_shell_class_part_rec;
      application_shell_class_part : application_shell_class_part_rec;
   end record;

   for application_shell_class_part use record at mod implementation_defined;
      core_class_part
      at 0
      range 0 .. core_class_part_size - 1;
      composite_class_part
         at 0
         range core_class_part_size .. composite_class_part_size - 1;
      shell_class_part
      at 0
      range composite_class_part_size .. shell_class_part_size - 1;
      override_shell_class_part
         at 0
         range shell_class_part_size .. override_shell_class_part_size - 1;
      vm_shell_class_part
         at 0
         range override_shell_class_part_size .. vm_shell_class_part_size - 1;
      vendor_shell_class_part
         at 0
         range vm_shell_class_part_size ..
               vendor_shell_class_part_size - 1;
```

```
      transient_shell_class_part
         at 0
         range vendor_shell_class_part_size ..
               transient_shell_class_part_size - 1;
      top_level_shell_class_part
         at 0
         range transient_shell_class_part_size ..
               top_level_shell_class_part_size - 1;
      application_shell_class_part
         at 0
         range top_level_shell_class_part_size ..
               application_shell_class_part_size - 1;
   end record;

   type application_shell_class_part_pointer is
                                    access application_shell_class_part_rec;
   type application_shell_class_pointer is
                                       access application_shell_class_part;

   -- allocate the class constant

   function to_widget_class is new unchecked_conversion(
      source => shell_class_pointer,
      target => widget_class);

   the_shell_class : constant widget_class :=
      to_widget_class (new shell_class_part);

   function to_widget_class is new unchecked_conversion(
      source => override_shell_class_pointer,
      target => widget_class);

   the_override_shell_class : constant widget_class :=
      to_widget_class (new override_shell_class_part);

   function to_widget_class is new unchecked_conversion(
      source => wm_shell_class_pointer,
      target => widget_class);

   the_wm_shell_class : constant widget_class :=
      to_widget_class (new wm_shell_class_part);

   function to_widget_class is new unchecked_conversion(
      source => vendor_shell_class_pointer,
      target => widget_class);

   the_vendor_shell_class : constant widget_class :=
      to_widget_class (new vendor_shell_class_part);

   function to_widget_class is new unchecked_conversion(
      source => transient_shell_class_pointer,
      target => widget_class);
```

```
   the_transient_shell_class : constant widget_class :=
      to_widget_class (new transient_shell_class_part);

   function to_widget_class is new unchecked_conversion(
      source => top_level_shell_class_pointer,
      target => widget_class);

   the_top_level_shell_class : constant widget_class :=
      to_widget_class (new top_level_shell_class_part);

   function to_widget_class is new unchecked_conversion(
      source => application_shell_class_pointer,
      target => widget_class);

   the_application_shell_class : constant widget_class :=
      to_widget_class (new application_shell_class_part);

end shell_private;
```

4.5 Pop-Up Widgets

The package specification *xt_geometry_management* defines the types and subprograms specified in chapter 6 of [3]. The semantics are unchanged.

```
package xt_geometry_management is

      type xt_geometry_result is
         (xt_geometry_yes,
          xt_geometry_no,
          xt_geometry_almost,
          xt_geometry_done);

      type xt_stack_mode is
         (xt_above,              --  Above in x_lib_.a Stack_Mode_Type
          xt_below,              --  Below in x_lib_.a Stack_Mode_Type
          xt_top_if,             --  Top_If in x_lib_.a Stack_Mode_Type
          xt_bottom_if,          --  Bottom_If in x_lib_.a Stack_Mode_Type
          xt_opposite,           --  Opposite in x_lib_.a Stack_Mode_Type
          xt_dont_change);       --  not in x_lib_.a

      subtype xt_geometry_mask is x_windows.boolean_array (0 .. 7);

      type xt_widget_geometry is record
         request_mode : xt_geometry_mask;
         x, y : position;
         width, height, border_width : dimension;
         sibling : widget;
         stack_mode : xt_stack_mode;
      end record;

      xt_null_geometry_mask : constant xt_geometry_mask :=
```

```
                                        xt_geometry_mask'(others => false);
    xt_cw_x : constant xt_geometry_mask :=
                      xt_geometry_mask'(0 => true, others => false);
    xt_cw_y : constant xt_geometry_mask :=
                      xt_geometry_mask'(1 => true, others => false);
    xt_cw_width : constant xt_geometry_mask :=
                      xt_geometry_mask'(2 => true, others => false);
    xt_cw_height : constant xt_geometry_mask :=
                      xt_geometry_mask'(3 => true, others => false);
    xt_cw_border_width : constant xt_geometry_mask :=
                      xt_geometry_mask'(4 => true, others => false);
    xt_cw_sibling : constant xt_geometry_mask :=
                      xt_geometry_mask'(5 => true, others => false);
    xt_cw_stack_mode : constant xt_geometry_mask :=
                      xt_geometry_mask'(6 => true, others => false);
    xt_cw_query_only : constant xt_geometry_mask :=
                      xt_geometry_mask'(7 => true, others => false);

  procedure xt_make_geometry_request
       (w              : widget;
        request        : in out xt_widget_geometry;
        reply_return   : in out xt_widget_geometry;
        result         : out xt_widget_geometry);

   procedure xt_make_resize_request
       (w : widget;
        width, height                : dimension;
        width_return, height_return : out dimension;
        result : out xt_geometry_result);

   procedure xt_move_widget (w    : widget;
                             x, y : position);

   procedure xt_resize_widget (w : widget;
                               width, height ; dimension;
                               border_width : dimension);

   procedure xt_configure_widget (w : widget;
                                  x, y : position;
                                  width, height : dimension;
                                  border_width : dimension);

   procedure xt_resize_window (w : widget);

   procedure xt_query_geometry
                       (w                 : widget;
                        intended          : xt_widget_geometry;
                        preferred_return  : out xt_widget_geometry;
                        result            : xt_geometry_result);

end xt_geometry_management;
```

Three procedure types are needed by geometry management. The first is the *resize* procedure which is of the previously defined type *xt_widget_proc*. The remaining two, *geometry_manager* and *query_geometry*, are of type *xt_geometry_handler*:

```
pseudo_type xt_geometry_handler is
                    new proc_type(request         : xt_widget_geometry;
                                  geometry_return : xt_widget_geometry)
                                          return xt_geometry_result;
```

4.6 Geometry Management

The package specification *xt_geometry_management* defines the types and subprograms specified in chapter 6 of [3]. The semantics are unchanged.

```
package xt_geometry_management is

   type xt_geometry_result is
      (xt_geometry_yes,
       xt_geometry_no,
       xt_geometry_almost,
       xt_geometry_done);

   type xt_stack_mode is
      (xt_above,              --  Above in x_lib_.a Stack_Mode_Type
       xt_below,              --  Below in x_lib_.a Stack_Mode_Type
       xt_top_if,             --  Top_If in x_lib_.a Stack_Mode_Type
       xt_bottom_if,          --  Bottom_If in x_lib_.a Stack_Mode_Type
       xt_opposite,           --  Opposite in x_lib_.a Stack_Mode_Type
       xt_dont_change);       --  not in x_lib_.a

   subtype xt_geometry_mask is x_windows.boolean_array (0 .. 7);

   type xt_widget_geometry is record
      request_mode : xt_geometry_mask;
      x, y : position;
      width, height, border_width : dimension;
      sibling : widget;
      stack_mode : xt_stack_mode;
   end record;

   xt_null_geometry_mask : constant xt_geometry_mask :=
                                       xt_geometry_mask'(others => false);
   xt_cw_x : constant xt_geometry_mask :=
                       xt_geometry_mask'(0 => true, others => false);
   xt_cw_y : constant xt_geometry_mask :=
                       xt_geometry_mask'(1 => true, others => false);
   xt_cw_width : constant xt_geometry_mask :=
                       xt_geometry_mask'(2 => true, others => false);
   xt_cw_height : constant xt_geometry_mask :=
                       xt_geometry_mask'(3 => true, others => false);
   xt_cw_border_width : constant xt_geometry_mask :=
```

```
                    xt_geometry_mask'(4 => true, others => false);
    xt_cw_sibling : constant xt_geometry_mask :=
                    xt_geometry_mask'(5 => true, others => false);
    xt_cw_stack_mode : constant xt_geometry_mask :=
                    xt_geometry_mask'(6 => true, others => false);
    xt_cw_query_only : constant xt_geometry_mask :=
                    xt_geometry_mask'(7 => true, others => false);

  procedure xt_make_geometry_request
      (w             : widget;
       request       : in out xt_widget_geometry;
       reply_return  : in out xt_widget_geometry;
       result        : out xt_widget_geometry);

   procedure xt_make_resize_request
       (w : widget;
        width, height                : dimension;
        width_return, height_return : out dimension;
        result : out xt_geometry_result);

   procedure xt_move_widget (w    : widget;
                             x, y : position);

   procedure xt_resize_widget (w : widget;
                               width, height ; dimension;
                               border_width : dimension);

   procedure xt_configure_widget (w : widget;
                                  x, y : position;
                                  width, height : dimension;
                                  border_width : dimension);

   procedure xt_resize_window (w : widget);

   procedure xt_query_geometry
                      (w                : widget;
                       intended         : xt_widget_geometry;
                       preferred_return : out xt_widget_geometry;
                       result           : xt_geometry_result);

end xt_geometry_management;
```

Three procedure types are needed by geometry management. The first is the *resize* procedure which is of the previously defined type *xt_widget_proc*. The remaining two, *geometry_manager* and *query_geometry*, are of type *xt_geometry_handler*.

```
  pseudo_type xt_geometry_handler is
                  new proc_type(request         : xt_widget_geometry;
                                geometry_return : xt_widget_geometry)
                                        return xt_geometry_result;
```

4.7 Event Management

The following package specification defines the event management types and subprograms with semantics as described in chapter 7 of [3].

```
package xt_event_management is

      type xt_event_table is private;
      type interval_type is implementation_defined;
      type device is implementation_defined;

      -- xt types
      subtype xt_input_mask is x_windows.boolean_array (1 .. 32);
      xt_im_xevent : constant xt_input_mask :=
         xt_input_mask'(1 => true,
                        others => false);
      xt_im_timer : constant xt_input_mask :=
         xt_input_mask'(2 => true,
                        others => false);
      xt_im_alternate_input : constant xt_input_mask :=
         xt_input_mask'(3 => true,
                        others => false);

      xt_input_read_mask : constant xt_input_mask :=
         xt_input_mask'(4 => true,
                        others => false);
      xt_input_write_mask : constant xt_input_mask :=
         xt_input_mask'(5 => true,
                        others => false);
      xt_input_except_mask : constant xt_input_mask :=
         xt_input_mask'(6 => true,
                        others => false);


      function xt_app_add_input (app_context : application_context;
                                 source      : device;
                                 condition   : xt_input_mask;
                                 proc        : xt_input_callback_proc;
                                 client_data : x_windows.caddr_t)
                                   return xt_input_id;

      function xt_app_add_timeout (app_context : application_context;
                                   interval    : interval_type;
                                   proc        : xt_timer_callback_proc;
                                   client_data : x_windows.caddr_t)
                                    return xt_interval_id;

      procedure xt_remove_timeout(timer : in xt_interval_id);

      procedure xt_add_grab (w : widget;
                             exclusive, spring_loaded : boolean);

      procedure xt_remove_grab (w : widget);
```

```
procedure xt_set_keyboard_focus (subtree, descendant : widget);

function xt_call_accept_focus (w : widget;
                               t : x_windows.time) return boolean;

function xt_app_pending
    (app_context : application_context) return xt_input_mask;

procedure xt_app_peek_event (app_context  : application_context;
                             event_return : out x_event;
                             event_found  : out boolean);

procedure xt_app_next_event (app_context  : application_context;
                             event_return : out x_event);

procedure xt_app_process_event (app_context : application_context;
                                mask        : xt_input_mask);

function xt_dispatch_event (event : x_event) return boolean;

procedure xt_app_main_loop (app_context : application_context);

procedure xt_set_sensitive (w         : widget;
                            sensitive : boolean);

function xt_is_sensitive (w : widget) return boolean;

procedure xt_app_add_work_proc(app_context : application_context;
                               proc        : xt_work_proc;
                               client_data : x_windows.caddr_t);

procedure xt_remove_work_proc (proc: xt_work_proc);

procedure xt_add_event_handler
                        (w             : widget;
                         an_event_mask : event_mask;
                         non_maskable  : boolean;
                         proc          : xt_event_handler_proc;
                         client_data   : x_windows.caddr_t);

procedure xt_remove_event_handler
                        (w             : widget;
                         an_event_mask : event_mask;
                         non_maskable  : boolean;
                         proc          : xt_event_handler_proc;
                         client_data   : x_windows.caddr_t);

procedure xt_add_raw_event_handler
                        (w             : widget;
                         an_event_mask : event_mask;
                         non_maskable  : boolean;
```

```
                                proc          : xt_event_handler_proc;
                                client_data   : x_windows.caddr_t);

      procedure xt_remove_raw_event_handler
                               (w             : widget;
                                an_event_mask : event_mask;
                                non_maskable  : boolean;
                                proc          : xt_event_handler_proc;
                                client_data   : x_windows.caddr_t);

      function xt_build_event_mask (w : widget) return event_mask;

   private

      implementation_derived

end xt_event_management;
```

The following procedures types are defined for event management:

```
   pseudo_type xt_input_callback_proc is
                        new proc_type(client_data : x_windows.caddr_t;
                                      source      : device;
                                      id          : xt_input_id);

   pseudo_type xt_timer_callback_proc is
                        new proc_type(client_data : x_windows.caddr_t;
                                      id          : xt_interval_id);
   pseudo_type xt_accept_focus_proc is
                        new proc_type(the_widget  : widget;
                                       the_time    : x_windows.time)
                                         return boolean;
   pseudo_type xt_work_proc is
                        new proc_type(client_data  : x_windows.caddr_t)
                                         return boolean;

   pseudo_type  xt_expose_proc is
                        new proc_type(the_widget  : in out widget;
                                      the_event   : x_event;
                                      the_region  : x_region);

   pseudo_type xt_event_handler_proc is
                    new proc_type(the_widget  : widget;
                                  client_data : x_windows.caddr_t;
                                  the_event   : x_windows.events.event);
```

4.8 Callbacks

The package specification *xt_callbacks* defines the types and subprograms associated with callbacks. The semantics of the subprograms are as described in chapter 8 of [3]. In the Ada specification objects of type *xt_callback_list* differ from the C specification in that lists

should not be null terminated. For this reason we have changed the C procedure names for *XtAddCallbacks* and *XtRemoveCallbacks* to *xt_add_callback_list* and *xt_remove_callback_list* respectively. The functionality of these procedures remains the same.

```
package xt_callbacks is
      type xt_callback_status is
         (xt_callback_no_list,
          xt_callback_has_none,
          xt_callback_has_some);

      type xt_callback_rec is record
           callback : xt_callback_proc;
           closure  : x_windows.caddr_t;
      end record;

      type xt_callback_list is array (natural range <>) of xt_callback_rec;
      type xt_callback_list_ptr is access xt_callback_list;


      procedure xt_add_callback (w             : widget;
                                 callback_name : string;
                                 callback      : xt_callback_proc;
                                 client_data   : x_windows.caddr_t);

      procedure xt_add_callback_list (w             : widget;
                                      callback_name : string;
                                      callbacks     : xt_callback_list);

      procedure xt_remove_callback (w             : widget;
                                    callback_name : string;
                                    callback      : xt_callback_proc;
                                    client_data   : x_windows.caddr_t);

      procedure xt_remove_callback_list (w             : widget;
                                         callback_name : string;
                                         callbacks     : xt_callback_list);

      procedure xt_remove_all_callbacks (w             : widget;
                                         callback_name : string);

      procedure xt_call_callbacks (w             : widget;
                                   callback_name : string;
                                   client_data   : x_windows.caddr_t);

      function xt_has_callbacks
          (w             : widget;
           callback_name : string) return xt_callback_status;

end xt_callbacks;
```

The procedure type *xt_callback_proc* is defined in the following package specification:

```
pseudo_type xt_callback_proc is
                new proc_type(the_widget  : in widget;
                              client_data : x_windows.caddr_t;
                              call_data   : x_windows.caddr_t);
```

4.9 Resource Management

4.9.1 Interface to Resources

A difficult problem arose in specifying the interface to argument lists for several of the resource management subprograms. The C model, lacking function overloading, creates lists of untyped data (lists of many different types stored as a single type). This can be done in Ada but at the cost of requiring application and widget programmers to do *unchecked_conversions* to the list type. A solution is to provide a generic package which provides functions to relieve programmers of the conversion task.

The generics solution is not problem free. In implementing the generics, we discovered that some compilers (TeleSoft) do not permit *unchecked_conversion* of unconstrained types, and retrieving values of unconstrained types may not be possible. Arrays in Ada (this is generally true with any unconstrained Ada type) may have additional bytes added to the array to provide indexing and size information. In retrieving data the intrinsics do not know the type of the data, and retrieve data based solely on location and size. The Telesoft compiler added three words to the front of unconstrained arrays (which are not counted in the *'length* attribute) making it impossible to retrieve the value without making assumptions about a specific compiler's handling of arrays.

The Ada/Xt implementation imposes the additional restriction: resource types which are unconstrained types such as arrays, and variant and discriminant records, are not supported. One unconstrained type, *string*, is required in Xt, but since the intrinsics know about *string* types a test for a resource of type *string* is made when retrieving a resource value. *String* types have separately defined subprograms for setting and retrieving values.

The solution also falls short in retrieving resource values because it requires the user to use *'address* to provide a memory address for storing the retrieved data. This allows the intrinsics to simply copy data (of an unknown Ada type) from the resource field to a user's local data space without needing to know about the underlying type. However, this is an unfortunate use of system-dependent programming, and is in fact a bit "unsafe."

Another approach to building argument lists might be to set resource values individually, but not have them take effect immediately. A new *activate* procedure could then initiate the changes to the actual widget resources. This approach fails because the design of Xt permits use of the *set_values_hook* function for setting subpart resource values. Individual setting of resource values requires changing the interface to the *set_values_hook* procedure which is not desirable.

The following package specifications define the generic interface to these untyped lists,

and subprograms for setting and retrieving *string* resource values.

```
package resource_values is
      subtype xt_arg_val isx_windows.caddr_t;
      type xt_arg is
         record
            name  : xt_string;
            value : xt_arg_val;
         end record;
      type arg_list is array (natural range <>) of xt_arg;
      type arg_list_ptr is access arg_list;

      type xt_convert_arg is
         record
            address_mode : xt_address_mode;
            address_id   : x_windows.caddr_t;
            size         : cardinal;
         end record;
     type xt_convert_arg_list is array (natural range <>) of xt_convert_arg;

     type xrm_value is
        record
           size : x_windows.x_integer;
           address : x_windows.caddr_t;
     end record;
     type xrm_value_ptr is access xrm_value;
     type xrm_value_ptr_list is array (natural range <>) of xrm_value_ptr;

      procedure xt_set_arg (arg   : in out xt_arg;
                            name  : in string;
                            value : in system.address);

      procedure xt_set_arg (arg   : in out xt_arg;
                            name  : in string;
                            value : in string);

      function set_convert_arg(mode : in xt_address_mode;
                               size : in cardinal;
                               res  : in system.address)
                                              return xt_convert_arg;

      function set_xrm_value(size : cardinal;
                             res : system.address) return xrm_value;

      function xt_merge_arg_lists (args1, args2 : arg_list) return arg_list;

     generic
         type resource_type is private;
         resource_size : in out cardinal;
     package resource_interface is
         procedure xt_set_arg(arg  : in out xt_arg;
                              name : in string;
                              res  : in resource_type);
```

```
        function set_convert_arg(mode : xt_address_mode;
                                 res  : resource_type)
                                        return xt_convert_arg;

        function set_convert_arg(position : in integer;
                                 res : resource_type)
                                        return xt_convert_arg;

        function init_xrm_resource(name   : in string;
                                   class  : in string;
                                   rtype  : in string;
                                   size   : in cardinal;
                                   offset : in cardinal;
                                   dtype  : in string;
                                   daddr  : in resource_type)
                                        return xrm_resource_ptr;

    end resource_interface;
end resource_values;
```

4.9.2 Representation of Resource Lists

Another major difference of the C and Ada specifications in resource management is the representation of resource lists. In C resource lists are initially represented as *XtResource* with string values for the various resource names. This is, in part, due to C's inability to execute conversion functions during aggregate array initialization. Ada can do the conversion to *XrmResource* during aggregate initialization. Furthermore, C does an in place conversion of the resource list in the widget class data structure which violates Ada's strong typing. As a result, the Ada specification defines resource lists to be of type *xrm_resource_list* in the widget class structure and provides conversion functions to create the proper lists. Several conversion functions are provided to allow conversion to the *xt_resource* type.

```
type xt_resource is
   record
      resource_name                  : xt_string;
      resource_class                 : xt_string;
      resource_type                  : xt_string;
      resource_size, resource_offset : cardinal;
      default_type                   : xt_string;
      default_address                : x_windows.caddr_t;
   end record;

type xt_resource_ptr is access xt_resource;

type xrm_resource is
   record
      resource_name                  : x_windows.xrm_name;
      resource_class                 : x_windows.xrm_class;
```

```
      resource_type                  : x_windows.xrm_quark;
      resource_size, resource_offset : cardinal;
      default_type                   : x_windows.xrm_quark;
      default_address                : x_windows.caddr_t;
   end record;
type xrm_resource_ptr is access xrm_resource;

type xt_resource_list is array (natural range <>) of xt_resource;
type xt_resource_list_ptr is access xt_resource_list;

type xrm_resource_list is
                  array (natural range <>) of xrm_resource_ptr;
type xrm_resource_list_ptr is access xrm_resource_list;

function create_xrm_resource (resource : xt_resource)
      return xrm_resource;

function create_xrm_resource_list
    (rlist : xt_resource_list) return xrm_resource_list_ptr;

function create_xt_resource_list
    (rlist : xrm_resource_list) return xt_resource_list_ptr;

function create_xt_resource
   (resource : xrm_resource) return xt_resource;

function xt_database
    (the_display : x_windows.display) return xrm_database;

procedure xt_get_resource_list
               (class           : widget_class;
                resources_return : out xt_resource_list_ptr);

procedure xrm_get_resource_list
               (class           : widget_class;
                resources_return : out xrm_resource_list_ptr);
```

4.9.3 Resource Management Package Specification

The package specification *xt_resource_management* defines the remaining resource management subprograms. These coincide with the definitions in chapter 9 of [3] except for the use of *xrm_resource_lists* instead of *xt_resource_lists*. The procedure *xt_get_resource_list* is overloaded to supply either the list in "quarked" form or in string form.

```
package xt_resource_management is

   type xt_address_mode is
      (xt_address,
       xt_base_offset,
       xt_immediate,
       xt_resource_string,
```

```
      xt_resource_quark);

type xt_convert_arg is
   record
      address_mode : xt_address_mode;
      address_id   : x_windows.caddr_t;
      size         : cardinal;
   end record;

procedure xt_get_subresources (w         : widget;
                               base      : x_windows.caddr_t;
                               name      : string;
                               class     : string;
                               resources : xrm_resource_list;
                               args      : arg_list);

procedure xt_get_application_resources (w         : widget;
                                        base      : x_windows.caddr_t;
                                        resources : xrm_resource_list;
                                        args      : arg_list);

procedure xt_string_conversion_warning (src, dst_type : string);

procedure xt_app_add_converter
               (app_context        : application_context;
                from_type, to_type : string;
                converter          : xt_converter_proc;
                converter_args     : xt_convert_arg_list);

procedure xt_convert (w         : widget;
                      from_type : string;
                      from      : xrm_value_ptr;
                      to_type   : string;
                      to        : out xrm_value_ptr);

procedure xt_direct_convert (converter : xt_converter_proc;
                             args      : xrm_value_ptr_list;
                             from      : xrm_value_ptr;
                             to_return : out xrm_value_ptr);

procedure xt_get_values (w    : widget;
                         args : arg_list);

procedure xt_get_subvalues (base      : x_windows.caddr_t;
                            resources : xrm_resource_list;
                            args      : arg_list);

procedure xt_set_values (w    : in out widget;
                         args : arg_list);

procedure xt_set_subvalues (base      : x_windows.caddr_t;
                            resources : xrm_resource_list;
```

```
                                    args      : arg_list);

end xt_resource_management;
```

The following procedures types are defined for resource management:

```
pseudo_type xt_args_proc is
             new proc_type(the_widget  : in out widget;
                           the_arglist : arg_list);

pseudo_type xt_set_value_func is
             new proc_type(current_widget : in widget;
                           request_widget : in widget;
                           new_widget     : in widget)
                                   return boolean;

pseudo_type xt_almost_proc is
   new proc_type(the_widget        : in out widget;
                 new_widget_return : out widget;
                 request, reply    : in out xt_widget_geometry);

pseudo_type xt_args_func is
             new proc_type(the_widget  : in widget;
                           the_arglist : arg_list) return boolean;

pseudo_type xt_resource_default_proc is
             new proc_type(the_widget  : in out widget;
                           offset      : in cardinal;
                           value       : in out xrm_value);

pseudo_type xt_converter_proc is
             new proc_type(args        : in xrm_value_list;
                           from        : in xrm_value;
                           to          : out xrm_value);
```

4.10 Translation Management

The translation management subprograms and data types are specified in the following package and retain the same semantics as specified in chapter 10 of [3].

```
package xt_translation_management is
   type xt_translations is implementation_defined;
   type xt_accelerators is implementation_defined;

   type xt_action_rec is
      record
         action_name : xt_string;
         action_proc : xt_action_proc;
      end record;
   type xt_action_list is array (natural range <>) of xt_action_rec;
   type xt_action_list_ptr is access xt_action_list;
```

```
procedure xt_app_add_actions (app_context : application_context;
                              actions     : xt_action_list);

function xt_parse_translation_table
    (table : string) return xt_translations;

procedure xt_augment_translations (w            : widget;
                                   translations : xt_translations);

procedure xt_override_translations (w            : widget;
                                    translations : xt_translations);

procedure uninstall_translations (w : widget);

function xt_parse_accelerator_table
    (table : string) return xt_accelerators;

procedure xt_install_accelerators (destination, source : widget);

procedure xt_install_all_accelerators (destination, source : widget);

procedure xt_set_key_translator (the_display : x_windows.display;
                                 proc        : xt_key_proc);

procedure xt_translate_keycode
    (the_display      : x_windows.display;
     the_keycode      : x_windows.keycode;
     some_modifiers   : modifiers;
     modifiers_return : out modifiers;
     keysym_return    : out x_windows.key_sym);

procedure xt_register_case_converter
    (the_display : x_windows.display;
     proc        : xt_case_proc;
     start, stop : x_windows.keyboard.key_sym);

procedure xt_convert_case
    (the_display                : x_windows.display;
     some_keysym                : x_windows.key_sym;
     lower_return, upper_return : out x_windows.key_sym);

end xt_translation_management;
```

The following procedure types are specified for translation management:

```
pseudo_type xt_action_proc is
            new proc_type(the_widget  : in out widget;
                          the_event   : in x_windows.x_event;
                          params      : in string;
                          num_params  : cardinal);

pseudo_type xt_string_proc is
            new proc_type(the_widget  : in out widget;
```

```
                                str          : xt_string);

    pseudo_type xt_key_proc is
        new proc_type(the_display      : x_windows.display;
                      the_keycode      : x_windows.keycode;
                      some_modifiers   : modifiers;
                      modifiers_return : out modifiers;
                      keysym_return    : out x_windows.key_sym);

    pseudo_type xt_case_proc is
        new proc_type(the_keysym   : in x_windows.key_sym;
                      lower_return : out x_windows.key_sym;
                      upper_return : out x_windows.key_sym);
```

4.11 Utility Functions

Some differences exist in the utility functions due to language differences. For example, the C function *XtNumber* is not needed in Ada because Ada provides the *'length* array attribute. The memory management functions may not be used in the same way as in a C implementation since Ada has *new* for allocating storage for its pointer types. For types the intrinsics does not know about (widgets), memory management functions are necessary. The memory management functions return an implementation defined type which should be some form of address to the allocated storage. The remaining subprograms provide the same functionality as defined in chapter 11 of [3].

The following types are assumed to be visible to the utilities package, and could be defined in this package:

```
type ptr is implementation_defined;
type string_list is array (natural range <>) of xt_string;
```

The following package defines the utility interfaces for Ada/Xt:

```
package xt_utilities is

   -- translating strings to widget instances:

   function xt_name_to_widget (reference : widget;
                               names     : string) return widget;

   -- managing memory usage:

   function xt_malloc (size : cardinal) return ptr;

   function xt_calloc (num, size : cardinal) return ptr;

   function xt_realloc (p   : ptr;
                        num : cardinal) return ptr;
```

```
procedure xt_free (p : in out ptr);

-- sharing graphics contexts:

function xt_get_gc (w          : widget;
                    value_mask : xt_gc_mask;
                    values     : x_windows.x_gc_values)
             return x_windows.gc;
-- is this list or array?

procedure xt_release_gc (w      : widget;
                         the_gc : x_windows.gc);

-- managing selections:

procedure xt_app_set_selection_timeout
                    (app_context : application_context;
                     timeout     : x_windows.time);

function xt_app_get_selection_timeout
    (app_context : application_context) return x_windows.time;

procedure xt_get_selection_value
    (w                 : widget;
     selection, target : x_windows.atom;
     callback          : xt_selection_callback_proc;
     client_data       : x_windows.caddr_t;
     timestamp         : x_windows.time);

procedure xt_get_selection_values
                        (w           : widget;
                         selection   : x_windows.atom;
                         targets     : x_windows.atom_list;
                         callback    : xt_selection_callback_proc;
                         client_data : x_windows.caddr_t;
                         timestamp   : x_windows.time);

function xt_own_selection
    (w               : widget;
     selection       : x_windows.atom;
     timestamp       : x_windows.time;
     convert_proc    : xt_convert_selection_proc;
     lose_selection  : xt_lose_selection_proc;
     done_proc       : xt_selection_done_proc) return boolean;

procedure xt_disown_selection(w : widget;
                              selection : x_windows.atom;
                              timestamp : x_windows.time);

-- merging events into a region

procedure xt_add_exposure_to_region
```

```
                          (event  : x_windows.event;
                           region : x_windows.region);

-- translating widget coordinates

procedure xt_translate_coords (w : widget;
                               x, y : position;
                               rootx_return : out position;
                               rooty_return : out position);

-- translating a window to a widget

function xt_window_to_widget
    (the_display : x_windows.display;
     the_window  : x_windows.window) return widget;

-- handling errors

function xt_app_get_error_database
    (app_context : application_context) return x_windows.xrm_database;

procedure xt_app_get_error_database_text
    (app_context          : application_context;
     name, restype, class : string;
     default              : string;
     buffer_return        : in out string;
     database             : x_windows.xrm_database);

procedure xt_app_set_error_msg_handler
    (app_context : application_context;
     msg_handler : xt_error_msg_handler_proc);

procedure xt_app_error_msg (app_context : application_context;
                            name, restype, class, default : string;
                            params : in out string_list);

procedure xt_app_set_warning_msg_handler
    (app_context : application_context;
     msg_handler : xt_error_msg_handler_proc);

procedure xt_app_warning_msg (app_context : application_context;
                              name, restype, class, default : string;
                              params : in out string_list);

procedure xt_app_set_error_handler
                        (app_context : application_context;
                         handler     : xt_error_handler_proc);

procedure xt_app_error (app_context : application_context;
                        message     : string);

procedure xt_app_set_warning_handler
```

```
                        (app_context : application_context;
                         handler     : xt_error_handler_proc);

   procedure xt_app_warning (app_context : application_context;
                             message     : string);

end xt_utilities;
```

The following procedure types are defined for use by the utilities:

```
   pseudo_type xt_convert_selection_proc is
       new proc_type(the_widget    : widget
                     selection     : x_windows.atom;
                     target        : x_windows.atom;
                     type_return   : x_windows.atom;
                     value_return  : out x_windows.caddr_t;
                     length_return : out cardinal;
                     format_return : out x_windows.x_integer)
                                          return boolean;

   pseudo_type xt_lose_selection_proc is
        new proc_type(the_widget  : in out widget;
                      selection   : in out x_windows.atom;
                      target      : in out x_windows.atom);

   pseudo_type xt_selection_callback_proc is
        new proc_type(the_widget     : in out widget;
                      client_data    : in x_windows_caddr_t;
                      selection      : in out x_windows.atom;
                      selection_type : in x_windows.atom;
                      value          : x_windows.caddr_t;
                      length         : in cardinal);

   pseudo_type xt_error_msg_handler_proc is
        new proc_type(resource_name  : string;
                      resource_type  : string;
                      resource_class : string;
                      default_p      : string;
                      params         : string_list);

   pseudo_type xt_error_handler_proc is
                       new proc_type(message : string);
```

5 Appendix A: Case-Statement Procedure Types

```
with system; use system;
package body callback_mechanism is

   type callback_mapped_id is
      range callback_id_range'first .. callback_id_range'last * NUM_CALLBACKS;

   unmapped_id: constant callback_mapped_id:= callback_mapped_id'first;

   callback_id_map : array(callback_id_range) of callback_mapped_id:=
      (others => unmapped_id);

   starting_at: callback_mapped_id:= callback_mapped_id'first + 1;
   next_mapped_id: callback_mapped_id:= starting_at;

   package body callback_ids is

      next_id: callback_id_range:= callback_id_range'first + 1;

      -- return a unique callback id
      function next_callback_id return callback_id_range is
         i: callback_id_range:= next_id;
      begin
         next_id:= next_id + 1;
         return i;
      exception
         when constraint_error =>
            raise CALLBACK_RANGE_ERROR;
      end next_callback_id;

      -- select the callback id from the callback object
      function to_callback_id_range(id: callback_id_type)
         return callback_id_range is
      begin
         return id.the_callback_id;
      end to_callback_id_range;

   end callback_ids;


   -- these procedures should never be called, so raise exception
   procedure default_next_call_back(id: callback_id_type; s: string) is
   begin
      raise CALLBACK_CALL_ERROR;
   end;
   procedure default_callback(s: string) is
   begin
      raise CALLBACK_CALL_ERROR;
   end default_callback;
```

```
package body callbacks is
   -- each instantiation of callbacks has a distinct id range
   low_range, high_range: callback_mapped_id:= callback_mapped_id'last;

   procedure callback (id : callback_id_type; s: string) is
      -- subtype assignment allows use of case statement
      subtype callback_range is callback_mapped_id range 1 .. NUM_CALLBACKS;
      mapped_id: callback_mapped_id:=
         callback_id_map(to_callback_id_range(id));
      index: callback_range;
   begin
      if mapped_id in low_range .. high_range then
         index:= mapped_id - low_range + 1;
         case index is
            when 1 => cb1(s); -- call the actual callback
            when 2 => cb2(s);
            when 3 => cb3(s);
         end case;
      else -- in the range of a previous instantiation
         next_callback(id, s);
      end if;
   end callback;

begin -- initialize
   low_range:= starting_at;
   high_range:= low_range + NUM_CALLBACKS - 1;
   starting_at:= high_range + 1;

   -- do this if .. then code for each formal callback
   if cb1'address /= default_callback'address then
      if id1 /= null_id then
         if callback_id_map(to_callback_id_range(id1)) /= unmapped_id then
            raise CALLBACK_INSTALL_ERROR; -- valid procedure, duplicate id
         else
            callback_id_map(to_callback_id_range(id1)):= next_mapped_id;
         end if;
      else
         raise CALLBACK_INSTALL_ERROR; -- valid procedure, null id
      end if;
   end if;
   next_mapped_id:= next_mapped_id + 1;

   if cb2'address /= default_callback'address then
      if id2 /= null_id then
         if callback_id_map(to_callback_id_range(id2)) /= unmapped_id then
            raise CALLBACK_INSTALL_ERROR; -- valid procedure, duplicate id
         else
            callback_id_map(to_callback_id_range(id2)):= next_mapped_id;
         end if;
      else
         raise CALLBACK_INSTALL_ERROR; -- valid procedure, null id
      end if;
```

```
      end if;
      next_mapped_id:= next_mapped_id + 1;

      if cb3'address /= default_callback'address then
         if id3 /= null_id then
            if callback_id_map(to_callback_id_range(id3)) /= unmapped_id then
               raise CALLBACK_INSTALL_ERROR; -- valid procedure, duplicate id
            else
               callback_id_map(to_callback_id_range(id3)):= next_mapped_id;
            end if;
         else
            raise CALLBACK_INSTALL_ERROR; -- valid procedure, null id
         end if;
      end if;
      next_mapped_id:= next_mapped_id + 1;

   end callbacks;

end callback_mechanism;
```

6 Appendix B: System-Dependent Procedure Types

```
package body xt_procedure_types is

package body xt_widget_class_procs is

   -- private data and functions

   bad_procedure_reference : exception;

   xt_inherit_constant_record : xt_widget_class_proc_rep;
   xt_inherit_constant        : xt_widget_class_proc;

   -- arg_record is a record encapsulation for arguments to ada
   -- functions.  Encapsulating as a record permits us to write
   -- exactly one foreign language dispatcher, which will call
   -- an intermediary Ada subprogram by pointer which will in turn
   -- call the Ada subprogram of an arbitrary parameter profile.

   type arg_record is record
      the_widget_class : widget_class;
      -- other parameters to the call function go here...
   end record;
   type arg_record_pointer is access arg_record;

   arg_record_buffer : arg_record; -- global arg_record for use as
                                   -- argument passing vehicle.  Could
                                   -- also make local to procedure call,
                                   -- but then would be on stack.  Is
                                   -- that better or worse?

   procedure default_xt_widget_class_proc(the_widget_class : widget_class) is
   begin
      raise bad_procedure_reference;
   end;


   -- type converters for converting to/from system.address

   function address_to_arg_record_pointer is
      new unchecked_conversion(
         source => system.address, target => arg_record_pointer);

   function xt_widget_class_proc_to_address is
      new unchecked_conversion(
         source => xt_widget_class_proc,
         target => system.address);

   -- visible functions

   function xt_inherit_widget_class_proc return xt_widget_class_proc is
   begin
```

```
      return xt_inherit_constant;
   end;

   procedure call(
      the_proc_id : xt_widget_class_proc; the_widget_class : widget_class) is
   begin

      -- construct the argument passing buffer
      arg_record_buffer.the_widget_class := the_widget_class;

      -- call the C or assembler or ... routine which calls the function
      dispatch_interfaces.call_ada(
         xt_widget_class_proc_to_address(the_proc_id),
         arg_record_buffer'address);
   end;

   package body procedure_pointer is

      -- procedure intermediary_caller is the actual subprogram which
      -- invokes the user supplied function.  The address of intermediary_caller
      -- is put into the id returned by the instantiation.

      procedure intermediary_caller(
         -- the_proc_id : xt_widget_class_proc;
         arg_record_address  : system.address) is

         an_arg_record_pointer : arg_record_pointer :=
            address_to_arg_record_pointer(arg_record_address);
      begin
         -- "the_proc" is the generic procedure
         the_proc(an_arg_record_pointer.the_widget_class);
      end;

   begin
      -- Vads version does not need to save context information
      -- for intermediary_caller
      declare
         temp_xt_widget_class_proc_rep : xt_widget_class_proc_rep;
      begin
         dispatch_interfaces.save_environment_context(
            temp_xt_widget_class_proc_rep'address);
         temp_xt_widget_class_proc_rep.proc_address :=
            intermediary_caller'address;
         proc_id :=
            new xt_widget_class_proc_rep'(temp_xt_widget_class_proc_rep);
      end;
   end procedure_pointer;

begin
   xt_inherit_constant_record.proc_address :=
      xt_inherit_widget_class_proc'address;
   xt_inherit_constant :=
```

```
      new xt_widget_class_proc_rep'(xt_inherit_constant_record);
end xt_widget_class_procs;

-- other procedure type packages...

end xt_procedure_types;


with system; use system;
package dispatch_interfaces is

-- Dispatch_interfaces provides the entries for assembler or C or ...
-- code which saves subrogram environment data and invokes Ada subprograms
-- via their address.

   procedure call_ada(
      the_proc_descriptor : system.address;
      the_arg_descriptor : system.address);

   pragma interface ( C, call_ada );

   procedure save_environment_context(the_proc_descriptor : system.address);

   pragma interface (C, save_environment_context);

end dispatch_interfaces;


/* the C code for VADS, TeleSoft, and Tartan, which invokes the subprograms
   via their addresses */

typedef void (*Proc)();
typedef struct _proc_descriptor {
      Proc p;
      /*
      ALSYS data fields
      int gd;
      int tcb;
      int profile;
      */
   } ProcDescriptorRec, *ProcDescriptor;

void call_ada(pd, arg)
ProcDescriptor pd;
char *arg;
{
   /* Alsys version of call_ada is assembler which restores subprogram
      environment context, and passes the argument via the data register d0.
      For VADS, etc., the call stack is used to pass arguments, and
      a simple JSR will suffice. */
   (*(pd->p))(arg);
}
```

```
void save_environment_context(pd)
ProcDescriptor pd;
{
   /* null body for VADS, TeleSoft, and Tartan. */
}
```

7 Appendix C: Simple Widget Definition

Simple Widget: Public Pseudo-Type

```
with intrinsics; use intrinsics;
with core_public; use core_public;
pragma elaborate(intrinsics);
package simple_public is

   -- 1) resources documentation:

   -- Name                 Class                RepType         Default Value
   -- ----                 -----                -------         -------------
   -- background           Background           Pixel           XtDefaultBackground
   -- border               BorderColor          Pixel           XtDefaultForeground
   -- borderWidth          BorderWidth          Dimension       1
   -- cursor               Cursor               Cursor          None
   -- destroyCallback      Callback             Pointer         NULL
   -- height               Height               Dimension       0
   -- insensitiveBorder    Insensitive          Pixmap          Gray
   -- mappedWhenManaged    MappedWhenManaged    Boolean         True
   -- sensitive            Sensitive            Boolean         True
   -- width                Width                Dimension       0
   -- x                    Position             Position        0
   -- y                    Position             Position        0


   -- 2) define constants for new resources.  Can we use enumeration and 'image?

   xt_n_cursor             : constant string := "cursor";
   xt_c_cursor             : constant string := "Cursor";
   xt_n_insensitive_border : constant string := "insensitive_border";
   xt_c_insensitive_border : constant string := "Insensitive";

   -- 3) define application interface to types and constants for intrisics use:

   subtype simple_widget is core_widget;
   subtype simple_class is core_class;

   function the_simple_class return simple_class;

-- 3a) define application type conversion operations


-- 4) define public entries to simple_widget operations:

   procedure foo(w : simple_widget); -- demonstration only

      -- NONE

end simple_public;
```

Simple Widget: Private Actual-Type

```
with intrinsics; use intrinsics;
with renamed_xlib_types; use renamed_xlib_types;
with x_windows;
with system;
with compiler_dependent;
with unchecked_conversion;

-- superclass context
with core_private; use core_private;

package simple_private is

   use xt_ancillary_types;
   use xt_procedure_types.xt_realize_procs;
   use xt_geometry_management;
   use xt_resource_management;
   use xt_translation_management;

   simple_part_rec_size : constant cardinal := 64;
   type simple_part_rec is record
      the_cursor : x_windows.cursors.cursor;
      insensitive_border : x_windows.pixmap;
   end record;

   simple_widget_size : constant cardinal :=
      core_part_size + simple_part_rec_size;
   type simple_widget_rec is record
      core_part : core_private.core_part;
      simple_part : simple_part_rec;
   end record;

   for simple_widget_rec use record at mod 2; -- alsys requires "2"
      core_part
      at 0
      range 0 .. core_part_size - 1;
      simple_part
      at 0
      range core_part_size .. core_part_size + simple_part_rec_size - 1;
   end record;

   type simple_part_pointer is access simple_part_rec;
   type simple_widget_pointer is access simple_widget_rec;

   simple_class_part_rec_size : constant cardinal := 32;
   type simple_class_part_rec is record
      is_change_sensitive : xt_realize_proc;
   end record;


   type simple_class_part is record
```

```
      core_class_part : core_private.core_class_part;
      simple_class_part : simple_class_part_rec;
   end record;

   for simple_class_part use record at mod 2;
      core_class_part
      at 0
      range 0 .. core_class_part_size - 1;
      simple_class_part
      at 0
      range core_class_part_size ..
            core_class_part_size + simple_class_part_rec_size - 1;
   end record;

   type simple_class_part_pointer is access simple_class_part_rec;
   type simple_class_pointer is access simple_class_part;

-- allocate the class constant

   function to_widget_class is new unchecked_conversion(
      source => simple_class_pointer,
      target => widget_class);

   the_simple_class : constant widget_class :=
      to_widget_class (new simple_class_part);

end simple_private;
```

Simple Widget: Public Pseudo-Type Implementation

```
-- superclass context
with core_public; use core_public;
with simple_private; use simple_private;

with x_windows; use x_windows;
with unchecked_conversion;
with system;
with renamed_xlib_types; use renamed_xlib_types;
with compiler_dependent;
with text_io;

pragma elaborate (simple_private);
pragma elaborate (core_public);
pragma elaborate (compiler_dependent);

pragma elaborate(text_io);

package body simple_public is

   use xt_ancillary_types;
   use xt_procedure_types.xt_widget_class_procs;
   use resource_manager;

-- type conversion operations:
   function to_simple_class is new unchecked_conversion(
      source => simple_class_pointer,
      target => simple_class);

   function to_simple_widget_pointer is new unchecked_conversion(
      source => simple_widget,
      target => simple_widget_pointer);

   function to_simple_class_pointer is new unchecked_conversion(
      source => widget_class,
      target => simple_class_pointer);

-- global objects:
   init_proc : xt_widget_class_proc;
   simple_class_constant : constant simple_class_pointer :=
      to_simple_class_pointer (simple_private.the_simple_class);


-- procedures for the class structure:
   procedure simple_class_part_initialize(wc : widget_class) is
   begin
      text_io.put_line("simple class class part initialization");
   end;


-- visibile operations:
```

```
   function the_simple_class return simple_class is
   begin
      return simple_class (simple_private.the_simple_class);
   end;


   procedure foo(w : simple_widget) is
      spp : simple_widget_pointer := to_simple_widget_pointer(w);
   begin
      spp.core_part.managed := xt_ancillary_types.xt_true;
   end;


begin

   declare
      -- procedure instantiations:
      package init_procs is new procedure_pointer(
      proc_id => init_proc,
      the_proc => simple_class_part_initialize );
   begin

      simple_class_constant.all := (
         core_class_part => (
            superclass => core_public.the_core_class,
            class_name => null,
            widget_size => simple_widget_size,
            class_initialize => null,
            class_part_initialize => init_proc, -- non-defaulted
            class_inited => xt_false,
            initialize => null,
            initialize_hook => null,
            realize => null,
            actions => null,
            resources=> null,
            the_xrm_class=> xrm_class'first,
            compress_motion => xt_false,
            compress_exposure => xt_false,
            compress_interleave => xt_false,
            visible_interest => xt_true,
            destroy => null,
            resize  => null,
            expose  => null,
            set_values => null,
            set_values_hook => null,
            set_values_almost => null,
            get_values_hook => null,
            accept_focus => null,  -- this should be *_func
            version => xt_version_type'first,
            callback_private => null,
            tm_table => null,
            query_geometry => null, -- should be *_func
```

```
            display_accelerator => null,
            extension => caddr_t(compiler_dependent.null_address) ),
         simple_class_part => (
            is_change_sensitive => null ));
   end;
end simple_public;
```

References

[1] Adele Goldberg and David Robson. *Smalltalk-80 The Language and Its Implementation.* Addison-Wesley, 1983.

[2] Hewlett-Packard. Programming with the xrlib user interface toolbox, 1988.

[3] Joel McCormack, Paul Asente, and Ralph R. Swick. X toolkit intrinsics - c language x interface, 1989. X Version 11, Release 3.

[4] National Institute of Standards and Technology. The user interface component of the applications portability profile, 1989. Draft FIPS.

[5] Andrew J. Palay, Fred Hansen, Mike Kazar, Mark Sherman, Maria Wadlow, Thomas Neuendorffer, Zalman Stern, Miles Bader, and Thom Peter. The andrew toolkit - an overview. In *Proceedings USENIX Technical Conference*, Winter 1988.

[6] Robert W. Scheifler and Jim Gettys. The x window system. *ACM Transactions on Graphics*, 5:79–109, April 1986.

[7] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[8] Kurt C. Wallnay and Robert Smith. Ada interfaces to x window system: Analysis and recommendations. Technical Report SDRL Q14-02021-D, STARS Technical Report, April 1989.